Pearson



Problem Solving with C++, Tenth Edition

C++入门经典

（第10版）

[美] Walter Savitch 著
周 靖 译

- Walter Savitch教授的C++入门经典，长销十几年
- 结构严谨、清晰，行文诙谐幽默，妙趣横生
- 更多技巧提示，更多生动有趣的编程实例
- 近70个精彩视频，410+自测题，230+编程项目

清华大学出版社



Problem Solving with C++，Tenth Edition

C++入门经典

（第10版）

[美] Walter Savitch 著
周 靖 译

清华大学出版社
北 京

内 容 简 介

作为 C++入门经典教材，本书结合作者多年的教学经验，清楚梳理出一个有利于教与学的结构体系，从各章开头的内容总览，到随处可见的自测题、小结框、编程提示和编程陷阱，再到章末的小结、习题、编程练习和编程项目，由浅入深，从简到繁，可以帮助初学者渐入佳境，逐步理解并掌握重要的编程概念。

全书共 18 章，8 个附录。在讲解 C++基础知识之后，循序渐进地引导读者深入函数、I/O 流、类、控制流程、命名空间、数组、字符串、指针和动态数组、递归、模板、指针和链表、派生类、异常以及标准模板库。本书前几版被全国 100 多所高校选作 C++程序设计课程的教材，很受师生欢迎。本书也适合自学，不同层次的知识点和测试练习，可以帮助读者以自己的节奏进入美妙的 C++编程世界。

图书在版编目(CIP)数据

C++入门经典：第 10 版/[美] 沃特 • 萨维奇(Walter Savitch)著；周靖译. —10 版. —北京：清华大学出版社，2018

书名原文：Problem Solving with C++(10th Edition)

ISBN 978-7-302-48676-3

Ⅰ. ①C… Ⅱ. ①沃… ②周… Ⅲ. ①C++语言—程序—设计—教材 Ⅳ. ①TP312.8

中国版本图书馆 CIP 数据核字(2017)第 269408 号

责任编辑：文开琪
封面设计：杨玉兰
责任校对：周剑云
责任印制：李红英
出版发行：清华大学出版社
网 址：http://www.tup.com.cn, http://www.wqbook.com
地 址：北京清华大学学研大厦 A 座 邮 编：100084
社 总 机：010-62770175 邮 购：010-62786544
投稿与读者服务：010-62776969, c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015, zhiliang@tup.tsinghua.edu.cn
印 装 者：北京密云胶印厂
经 销：全国新华书店
开 本：185mm×260mm **印 张**：47.75 **字 数**：993 千字
版 次：2018 年 1 月第 1 版 **印 次**：2018 年 1 月第 1 次印刷
定 价：138.00 元

产品编号：075771-01

译者序：大道至简

最近在看《罗素回忆录：来自记忆的肖像》，罗素的行文向来字字珠玑，具有很强的感染力，很容易引起读者的共鸣，能给读者带来许多启发。其中有一篇小文介绍他是如何写作的。他在文中讲到：“我希望用最少量的词便能够把每件事情说得一清二楚。我肯花时间设法找出最简洁的方式把某些事情毫不含糊地表达出来，为此，往往不惜牺牲追求美学上优点的一切企图。”在他 21 岁之前，希望自己的写作风格能够接近于约翰·米尔的风格，因为后者有值得他效仿的句型结构和拓展主题的方式。经过种种尝试之后，罗素终于醒悟，意识到对华丽词藻和张扬写作风格的模仿会诱发一定程度的虚伪性，认识到所有的模仿都是危险的，从而总结出一些简单的写作准则：

其一，如果可以使用一个简单的词，就永远不要使用一个复杂的词。其二，如果你想要做一个包含大量必要条件在内的说明，那么尽量把这些必要条件放在不同的句子里分别说清楚。其三，不要让句子的开头导致读者走向一个与结尾有抵触的结论。

由此联想到我们的教材，一本优秀的教材，其表述方式和语言应该能够使大家都能明白，而不是充斥着只有少数博学之士才能看懂的行话或术语。

在这次翻译并修订 Savitch 老教授的《C++入门经典》的过程中，有颇多这样的感受。真正的大师，是不会一味追求形式化、科学化和精致化，使得专业知识与普通读者渐行渐远，直到彼此之间竖起不可逾越的篱笆。真正优秀的作者，是不会把那些普通读者也能明白的事情说得高深莫测，让大家云里雾里的。Savitch 老先生的这本书让人感受到何为真正的大师，何为真正的优秀作者。这本书从 1995 年首次出版以来，经过二十年的考验，其通俗易懂、妙趣横生、与时俱进的特色，深受广大读者喜爱，被誉为“C++入门经典”，“C++入门教材的‘常青树’”。现在立足于 IT 行业的很多精英和骨干，很多都是在他的熏陶下成长起来的。本书目前已经修订到第 9 版，每次修订，都能增加新的特色，能体现时代的教学特征。

《C++入门经典》之所以畅销不衰，与其鲜明的特色是分不开的。

第一，文风朴实，循序渐进，可读性强(对于一本面向 C++初学者的书来说，这是最起码的要求)，而不像某些作者的书，思维跳跃得很厉害，读起来很费劲。大家知道，第二次世界大战期间，丘吉尔的演讲和文章最受欢迎，其中一个重要因素就是他善于用普通老百姓也能懂的浅显语言来阐述自己的观点。Savitch 老先生也是这样行文的。他考虑到广大读者的需求，广泛采用浅显易懂的语言来讲授 C++编程知识。为保持这一特色，我在翻译过程中，也尽量如此。无论原著还是译本，宗旨都是循循善诱地引导一个完全不懂 C++的人在短时间里充分熟悉并掌握 C++编程技术。稍微翻阅几页正文，您就能充分体会到这个特点。

第二，本书完全符合标准。本书的示范程序不仅完全符合最新的 ANSI/ISO C++标准，

还遵循行业通行的编程风格，这便于读者在学习本书之后写出任何程序员都可以理解、任何编译器都能通过的程序。

第三个特色体现在本书的组织和结构上。时下流行的一个观点是，学习 C++应该先从类学起。没问题，本书在创作时便充分考虑到了这一部分读者的要求。事实上，您可以按照自己喜欢的任何顺序来阅读各章的内容。这方面的详情可参考前言所提供的“依赖图”。由于这是一本真正的教科书，所以每章都提供了丰富的、重点突出的、非常有趣的自测题和编程项目。

第四，编程实例和编程项目贴近生活。文中几十个实例都源自生活，比如，信用卡余额、州收入所得税、购买比萨、温度换算、超市定价系统、回文测试、体重指数等，这些都能让我们体会到编程的乐趣。

第五，第 10 版新增了一些习题和视频讲解，并重点讨论了 C++11 的特色功能。借助约 70 个视频讲解，可进一步了解解题思路，牢固掌握基础知识。

一本好书，凝聚着作者很多心血。一本好的译作又何尝不是呢？不仅要仔细揣摩作者的意思，还必须在不篡改作者本意的基础上用通俗易懂的文字表达出来。译本如原作，此为“信”；文字通达，令国人一目了然，此为“达”；提炼文字，使之有文学价值，此为“雅”。

计算机书籍虽然不是文学作品，但随着读者水平的提高，也对“雅”提出了新要求。人们之所以需要这方面的译本，不仅是为了快速消化和吸收国外的最新技术和观点，还为了满足自己的阅读需求。所以对于译本来说，“信”和“达”固然重要，但“雅”也必不可少。这三者之间的关系是：只有做到“信”，才有可能进一步追求“达”，进而实现“雅”。

为确保正确性，我在翻译之余把书中的示范程序全部“跑”了一遍。根据我以前的经验，每本书都有这样或那样的错误，有的书错误之多，以至于最终还要出版数十页的勘误表。但是本书的测试结果令人惊叹，它的代码具有很高的正确性，这是一些同类教科书无法媲美的。与此同时，为保证正确性，我在翻译时除了参考网上公开的原书勘误，还和原作者进行了积极而卓有成效的沟通，对原著进行的所有改动都是经过授权的。这一过程有效确保了本书简体中文版的代码质量，使之达到了百分之百的正确率。

关于术语，本书简体中文版采用了业内通行的、国内程序员非常熟悉的词汇。基于翻译的本质，我的工作是让读者无障碍地阅读文档，并积极地跟上作者的思路。假如因为某个或者许多稀奇古怪的词而阻碍了阅读，那就是翻译工作的失败，这不是您我希望见到的。

本书第 4 版在国内出版以来，普遍受到读者的关注和欢迎。读者朋友们普遍反映喜欢此书的写作和翻译风格，喜欢穿插于全书的编程提示与陷阱，更喜欢书中难度不一的自测题和编程项目。有很多读者来信索取编程项目的答案。我也亲自做了部分有挑战性的编程项目，这的确是全书的特色和精华。由于这部分答案仅提供给教师，所以我建议大家发邮件到 coo@netease.com 申请，有人为您提供帮助。

从第 5 版开始，教材特色更加明显。例如，在各章开篇处，提供了能引起读者关注的小节标题(即当前章的大纲)，目的是让读者更清楚地理清脉络。新增一章内容专门介绍 STL(标准模板库)。自第 6 版开始，基本沿用了第 5 版的大纲，修订了部分内容，对章节进行了更合理的调整。各章开始处提供了饶有趣味的引文。这些引文要么出自计算机业界大师，权威、有见地；要么出自名著，巧妙运用双关语把互不关联的两种含义结合起来，既

诙谐有趣，又引人深思。

最后，因为再次重温《星际穿越》，我想借用老教授所念的几句诗与大家共勉：

Do not go gentle into that good night,

Old age should burn and rave at close of day;

Rage, rage against the dying of the light.

不要温顺地走入那个良夜

昏暮降临，垂老之人也要燃烧，也要喊

怒斥，怒斥那光的消减

Though wise men at their end know dark is right,

Because their words had forked no lightning they

Do not go gentle into that good night.

智者将逝，早知暗黑之必然

因其言渐微，茶也凉

然，他们绝不愿温顺地走入那个良夜

最后之最后，感谢原作者 Walter Savitch，是他写出了这样一本极其出色的 C++编程教材。感谢翻译过程中所涉及的所有人士，他们是文天山、成荣静、文瑞、刁灿群、黄洁、蒋昌友、李建、刘其星、刘琼、刘勇、江卫、莫永红、施玉梅、叶昌元、游美波、张攀飞和周建超。还要感谢女儿子衿，人生的道路充满选择，很欣慰她做出了自己的选择并勇敢地走在路上。

同时，也要感谢各位读者对我的支持和信赖，尤其是本书既往版本的读者，他们的反馈和支持让我倍感温暖，同时也激励和鞭策着我尽心尽力，奉献更好的书。更要感谢对本书提出意见和建议的朋友，例如清华大学毕业生张元章和清华大学在校博士生段菲，他们的“火眼金睛”进一步提高了本书的品质。感谢各位，我愿意尽自己的微薄之力，继续做好《C++入门经典》后续版本的翻译和维护工作。

再次祝愿读者朋友能够通过本书，开始愉快而激动人心的 C++编程之旅！

前　　言

本书适合 C++程序设计和计算机科学入门课程。不要求读者有任何编程经验，也不要求掌握除中学代数之外的其他任何数学知识。

本书前几版的读者请阅读关于第 10 版修订内容的小节，前言其余内容可略过。新读者请阅读前言的全部内容以把握本书脉络。

第 10 版修订内容

第 10 版采用和第 9 版相同的体例。保留第 9 版全部内容，但进行了以下修订。

- 统一采用 camelCase 拼写法，不再采用 C 风格的下划线连接。例如，`set_name` 改为 `setName`。
- 第 10 章讨论了浅拷贝和深拷贝。
- 第 12 章和第 17 章讨论了如何用头文件引入编译模板。
- 第 18 章讨论了 C++11 的 `std::array` 类、正则表达式、线程和智能指针。
- 进行了大量勘误和编辑，强调了文件 I/O 的首选方法，术语更规范，更好地定义了封装，并不再讨论已在 C++11 和更高版本中成为标准的一些东西。
- 新增 10 个编程项目。
- 新增 5 个视频讲解，总数达 69 个。这些视频讲解辅导学生解题和写程序，有助于巩固对关键编程概念的掌握。如书中某个主题有对应的视频讲解，就会出现一行特殊的注释(视频讲解：……)。

用过第 9 版的教师可沿用以前的教案，几乎不需要任何改动。

主题可以灵活排序

本书允许教师自由安排教学顺序。为了演示这一灵活性，下面推荐了两个额外的顺序。采用任何顺序都不会影响学习的连贯性。为了在改变顺序时确保这种连贯性，可能需要移动个别小节而不是全章。但只有较大的、位置便利的小节才需移动。为了帮助您根据需要自定义一个教学/阅读顺序，图 P.1 展示了一幅依赖图。另外，每章都有“预备知识”小节，解释学习那一章的每一节之前需掌握的内容。

重新排序 1：提前学习类

为有效设计类，学生需要掌握一些基本工具，比如控制结构和函数定义。这些基础知识在第 1 章～第 6 章介绍。完成第 6 章后，学生就可开始写自己的类了。如果想提前学习类的相关知识，可像下面这样重新安排各章顺序。

- **基础知识**　第 1 章～第 6 章。这 6 章全面介绍控制结构、函数定义和基本文件 I/O。第 3 章介绍几种额外的控制结构，想提前学习类可考虑推迟这一章。

- **类和命名空间** 第 10 章、第 11 章的 11.1 节和 11.2 节、第 12 章。这些章节全面介绍了如何定义类、友元、重载操作符和命名空间。
- **数组、字符串和向量** 第 7 章和第 8 章。
- **指针和动态数组** 第 9 章。
- **类中的数组** 第 11 章的 11.3 节和 11.4 节。
- **继承** 第 15 章。
- **递归** 第 14 章(可推迟)。
- **指针和链表** 第 13 章。

可能还要用到以下各章的部分内容。

- **异常处理** 第 16 章。
- **模板** 第 17 章。
- **标准模板库** 第 18 章。

重新排序 2：略微推迟类的学习

在“重新排序 2”中，将先学完所有控制结构和数组的知识，然后才开始学习类。虽然对类的接触要比“重新排序 1”晚，但还是比本书的默认顺序提前一些。

- **基础知识** 第 1 章到第 6 章。这 6 章全面介绍了控制结构、函数定义和基本文件 I/O。
- **数组和字符串** 第 7 章、第 8 章的 8.1 节和 8.2 节。
- **类和命名空间** 第 10 章、第 11 章的 11.1 节、11.2 节和第 12 章。这些章节全面介绍了如何定义类、友元、重载操作符和命名空间。
- **指针和动态数组** 第 9 章。
- **类中的数组** 第 11 章的 11.3 节和 11.4 节。
- **继承** 第 15 章。
- **递归** 第 14 章(可推迟)。
- **向量** 8.3 节。
- **指针和链表** 第 13 章。

可能还要用到以下各章的部分内容。

- **异常处理** 第 16 章。
- **模板** 第 17 章。
- **标准模板库** 第 18 章。

面向学生的易用性

一本书必须按恰当的顺序来讲解恰当的主题，这是最起码的要求。另外，在老师和其他有经验的程序员看来，书的内容必须清晰而正确，这是另一个最起码的要求。但是不是符合这两项要求的书都是好书呢？答案是否定的。书中的内容必须采取有利于初学者使用的方式来编排。在这本入门教科书中，我尽力让学生觉得清楚和友好。本书以前版本的大量学生反馈证明，这种写作风格确实使内容更清晰，能使学生充分享受到学习的乐趣。

ANSI/ISO C++标准

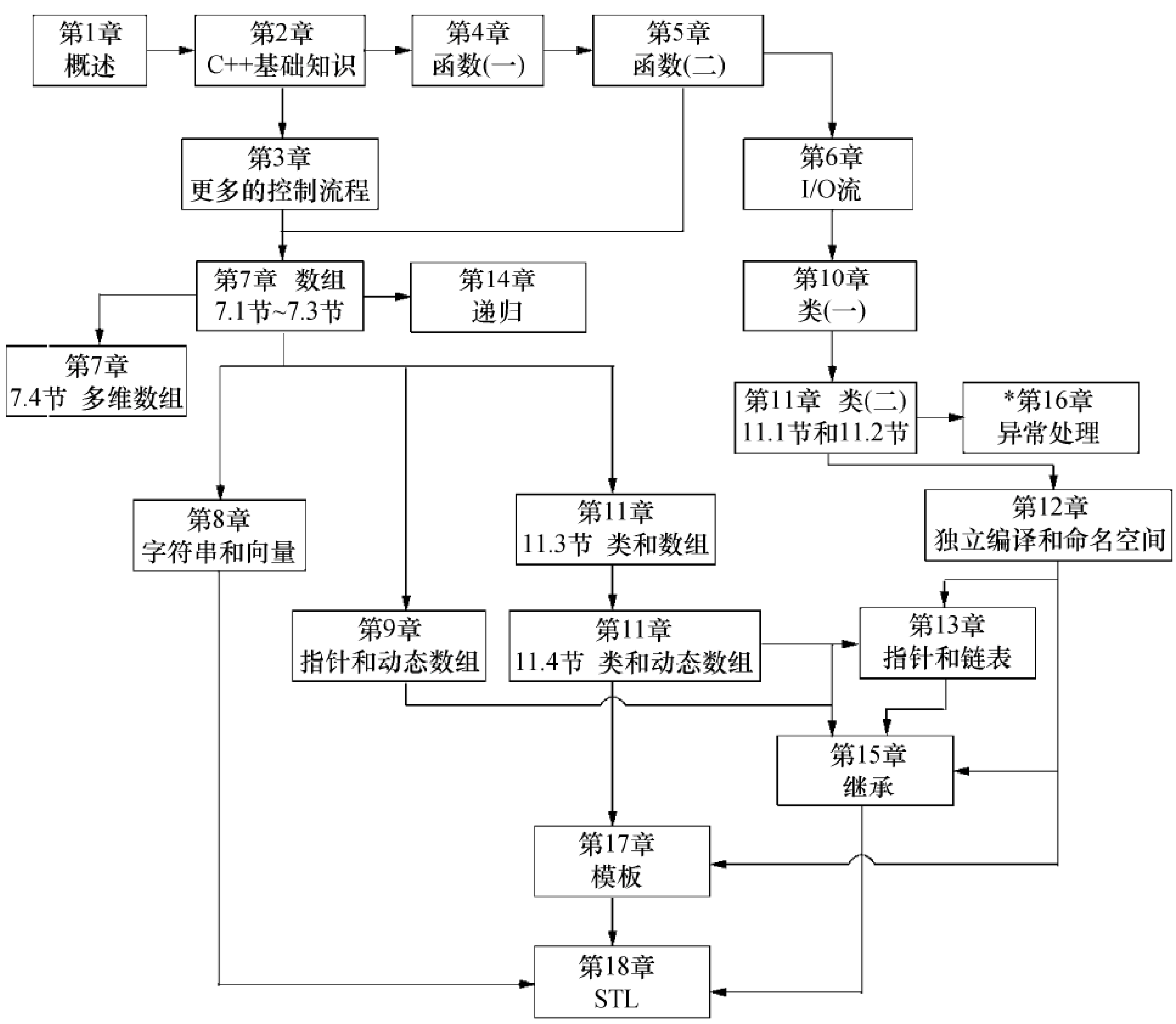
本书可顺利使用符合最新 ANSI/ISO C++标准的编译器。写作时最新标准是 C++14。

高级主题

许多“高级主题”都已成为标准 CS1 课程的一部分。即使不是，以补充材料的形式提供也不错。本书提供大量高级主题，它们既可集成到课程中，也可作为自学主题。本书全面讲述了 C++模板、继承(包括虚函数)、异常处理和 STL(Standard Template Library，标准模板库)。虽然本书使用了库，而且教给学生库的重要性，但不要求任何非标准库。本书只用所有 C++实现都有的库。

依赖图

下面的依赖图展示了各个章节可能的排序方式。连接两个框的实线表明上部的框必须先于下部的框完成。只要符合这个条件，采用任何阅读顺序都无损连贯性。如果一个框中包含小节编号，表明该框只代表那些小节，不代表全章。



* 第16章 偶而会提到派生类，但这些内容是可以忽略的

小结框

每个要点都用一个有底纹的方框来小结，它们散布于各章。

自测题

每章都在重要位置提供大量自测题。答案在章末提供。

视频讲解

视频讲解(Video Note)旨在讲解关键编程概念和技术，演示了从设计到编码来解决问题的过程。视频讲解使学生能方便地自学感兴趣的主题，支持选择、播放、倒退、快进和暂停。每当看到“视频讲解：……”，都表明当前主题有对应的视频讲解。视频列表请从本书中文版配套网站获取，网址是 *http://transbot.ys168.com* 和 *https://pan.baidu.com/s/1yd43W*。注意，由于是英文视频，所以为了方便索引，书中保留了这些视频的英文名称。

支持材料

部分支持材料本书所有读者都适合。其他仅适合认证教师。

适合本书所有读者的支持材料

- 源代码
- PowerPoint 幻灯片
- 视频讲解

获取这些材料请访问本书译者博客(*transbot.blog.163.com*)或网盘（*http://transbot.ys168.com* 和 *https://pan.baidu.com/s/1yd43W*）。

适合认证教师的资源

选用本书作为教材的教师，可致函责编信箱 *coo@netease.com*，了解详情。

- 教师资源指南(Instructor's Resource Guide)：包括每一章的教学要点、课堂测验/答案和大量编程项目的参考答案。
- Test Bank 和 Test Generator：用于生成试卷。
- PowerPoint 幻灯片：包括本书的程序和插图。
- Lab Manual(实验手册)。

致　　谢

在准备本书素材期间，许多个人和团体提供了大量建议和帮助，并和我进行了广泛而深入的探讨。本书第 1 版大部分内容是我在科罗拉多大学(波德分校)计算机系担任客座教授时写就的。其余内容和后续版本则是在加州大学(圣地亚哥分校)计算机科学与工程系完成的。非常感谢两所学校为本书写作和教学所提供的良好氛围。

特别感谢向这一版和之前各版提供意见或者编程项目的个人，以下人名按字母排序：Alex Feldman，Amber Settle，Andrew Burt，Andrew Haas，Anne Marchant，Barney MacCabe，Bob Holloway，Bob Matthews，Brian R. King，Bruce Johnston，Carol Roberts，Charles Dowling，Claire Bono，Cynthia Martincic，David Feinstein，David Teague，Dennis Heckman，Donald Needham，Doug Cosman，Dung Nguyen，Edward Carr，Eitan M. Gurari，Ethan Munson，Firooz Khosraviyani，Frank Moore，Gilliean Lee，Huzefa Kagdi，James Stepleton，Jeff Roach，Jeffrey Watson，Jennifer Perkins，Jerry Weltman，Joe Faletti，Joel Cohen，John J. Westman，John Marsaglia，John Russo，Joseph Allen，Joseph D. Oldham，Jerrold Grossman，Jesse Morehouse，Karla Chaveau，Ken Rockwood，Larry Johnson，Len Garrett，Linda F. Wilson，Mal Gunasekera，Marianne Lepp，Matt Johnson，Michael Keenan，Michael Main，Michal Sramka，Naomi Shapiro，Nat Martin，Noah Aydin，Nisar Hundewale，Paul J. Kaiser，Paul Kube，Paulo Franca，Richard Borie，Scot Drysdale，Scott Strong，Sheila Foster，Steve Mahaney，Susanne Sherba，Thomas Judson，Walter A. Manrique，Wei Lian Chen 和 Wojciech Komornicki。

还要感谢用过本书前几版的许多老师，他们的建议相当有价值。最后感谢 Kenrick Mock 帮我实现了这一版的全部修订。他的工作令我和编辑激赏！

目　　录

第 1 章　计算机和 C++编程入门

分析的整个发展和运作现在都可由机器完成……分析机将指引未来的科学发展。

——查尔斯·巴贝奇(1792—1871)

概述

本章讲解计算机的基本组成，以及设计和编写程序的基本技术，然后展示一个示例 C++程序，描述它是如何工作的。

1.1　计算机系统

计算机要遵循的一系列指令统称为**程序**。计算机使用的各种程序称为该计算机的**软件**。组装一台计算机所需的物理设备称为**硬件**。正如后文所述，计算机硬件在概念上是非常简单的。然而，现在的计算机都配备大量软件以辅助我们完成各种编程任务。这些软件包括各种编辑器、翻译器以及管理器等。最终的工作环境就是一个复杂的、功能强大的系统。本书几乎完全围绕软件展开，但首先有必要对硬件有一个基本的了解。

硬件

计算机主要分为 PC、工作站和大型主机。**PC**(个人电脑)是体积较小的计算机，设计目的为每次由一个人使用。大多数家用电脑都是 PC，但 PC 也广泛应用于商业、工业和科学领域。**工作站**其实是一台体积更大、功能更强的 PC。可把它视为一种“工业”PC。**大型主机**则是更大的计算机，通常要求一组支持人员，而且要供多个用户共享。PC、工作站和大型主机并不是泾渭分明的，但利用这些术语，通常能表达与一台计算机有关的常规信息。

网络由大量相互连接的计算机构成，以便这些计算机共享资源(比如打印机)和信息。一个网络可能包含大量工作站以及一台或多台大型主机，另外还有打印机之类的共享设备。

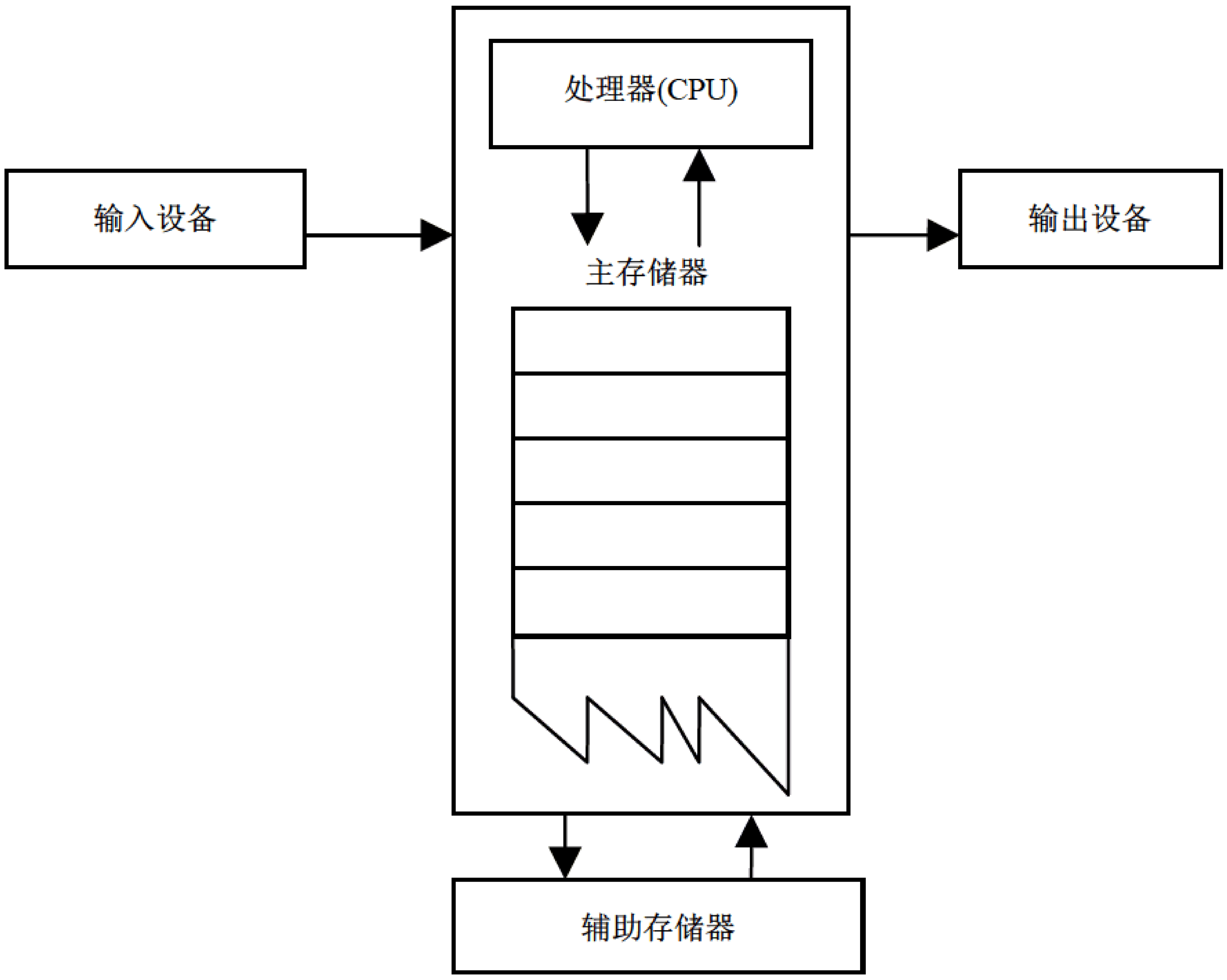
由于本书的目的是学习编程，所以无论使用 PC、大型主机还是工作站，都是无关紧要的。稍后就知道，这三种计算机的基本结构是相同的。

大多数计算机系统的硬件都像图 1.1 那样配置。计算机由 5 个主要部件构成：输入设备、输出设备、处理器(也称为 CPU)、主存储器以及辅助存储器。处理器、主存储器和辅助存储器通常安装到一个机箱内部。处理器和主存储器是计算机的核心，可将其视为一个集成单元。其他部件与主存储器相连，并遵照处理器的指示工作。图 1.1 中的箭头指明信息流动的方向。

输入设备是允许用户将信息发送给计算机的设备。主要的输入设备是键盘和鼠标。

输出设备是允许计算机将信息发送给用户的设备。最常用的输出设备是显示器，或者称为监视器。计算机系统一般有多个输出设备。例如，除了显示器之外，计算机还可能连接了一台打印机，能在纸张上输出。有时将键盘和显示器统称为**终端**。

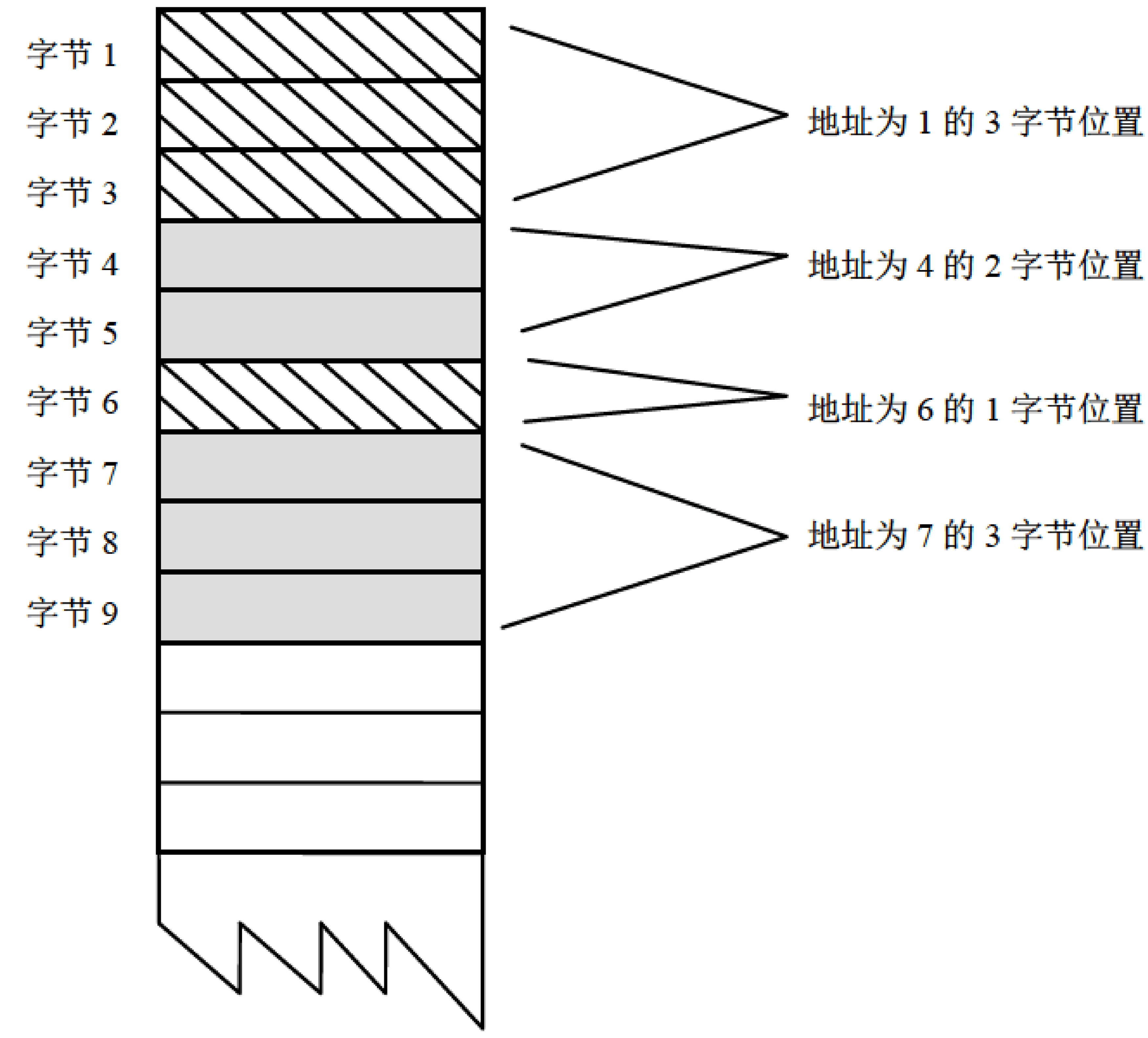
图 1.1　计算机的主要部件



为了存储输入并像平时那样用草稿纸来演算，计算机提供了**存储器**。计算机要执行的程序也存储在存储器中。计算机支持两种形式的存储器，分别为主存储器和辅助存储器。要执行的程序存储在主存储器中。正如主存储器这个名称所暗示的，它是最重要的存储器。**主存储器**相当于一个很长的编号位置列表，这些位置称为**存储位置**或者**内存位置**。在不同计算机中，内存位置的数量也是不同的，从几千到几百万都有，有的甚至能达到几十亿。每个内存位置都包含一系列 0 和 1。这些位置的内容可以改变。所以，每个内存位置都可被视为一块小黑板，计算机可在上面擦写。在大多数计算机中，每个内存位置的 0 或 1 个数相同。只能包含 0 或 1 的数位称为一个**二进制位**，或者称为**位**或**比特**。大多数计算机的内存位置都包含 8 位(或 8 位的倍数)。每 8 位称为一个**字节**，所以可以将这些编号内存位置称为字节。换言之，可将计算机的主存储器视为一个很长的编号存储单元(字节)列表。对字节进行标识的编号称为该字节的**地址**。一个数据项(数字或字母)可存储到其中一个字节中。以后需要该数据项时，就根据那个字节的地址来查找数据项。

如果要处理的数据项太大(比如一个很大的数)，以至于一个字节容不下，就使用几个相邻的字节容纳它。在这种情况下，用于容纳该数据项的整个内存块仍然称为一个**内存位置**。构成这个内存位置的第一个字节的地址作为这个较大的内存位置的地址。所以，一种更准确的说法是：可将计算机的主存储器视为一个很长的内存位置列表，每个位置长度可变。每个位置的长度用字节数来表示，第一个字节的地址成为该位置的地址。图 1.2 展示了一台假想计算机的主存储器。每个内存位置的长度都不是固定的，在计算机上运行一个新程序时，它们可能发生改变。

图 1.2　内存位置和字节



字节和地址

主内存被划分成称为**字节**的多个编号位置，一个字节的编号就是该字节的**地址**。一组连续的字节可作为一个数据项(比如数字或字母)的存储位置。组内第一个字节的位置就是这个更大的存储位置的地址。

在计算机存储器中，虽然信息实际表示成 0 和 1，但用 C++(或其他大多数编程语言)编程时，不必过于关心这个事实。不过，一旦开始写程序，许多人仍然希望知道 0 和 1 具体是如何使用和转换的。计算机必须将这些 0,1 序列解释成字母、数字、指令或者其他类型的信息。计算机根据特定的编码方案来自动执行这些解释。存储在计算机存储器中的每种类型的数据项都要采用一种不同的编码：字母使用一种代码，整数使用另一种代码，小数使用另一种代码，指令使用另一种代码，依此类推。例如，在一个常用的代码集中，01000001 是字母 A 的编码，也是数字 65 的编码。为了确定特定位置中的 01000001 代表的是什么，计算机必须跟踪记录目前那个位置目前使用的是哪一种编码。幸好，程序员很少需要关心这些编码，并可放心地假定位置中包含了实际的字母、数字或者其他数据项。

为什么是 8

每个**字节**都代表一个能容纳 8 个二进制位的内存位置。那么，8 有何特别之处？有两个原因使 8 显得很特殊。首先，8 是 2 的 3 次方。由于计算机在最底层使用的是二进制位，而每一位只有两个可能的值，所以 2 的乘方用起来比 10 的乘方更方便。其次，需要 8 位(1 个字节)才能对一个字符(比如一个英语字母或其他键盘符号)进行编码。

到目前为止，我们讨论的都是主存储器。没有主存储器，计算机不能做任何事情。但是，只有计算机真正按照一个程序中的指令工作时，才会用到主存储器。计算机还有另一种形式的存储器，称为**辅助存储器**或**辅助存储**。辅助存储器能在计算机使用之后(和之前)持久性地保存数据。还可以将辅助存储器称为辅助存储设备、外部存储器或者外部存储。

在辅助存储设备中，信息以**文件**为单位来保存。文件可大可小。例如，程序平时存储在辅助存储设备上的文件中，并在程序运行时复制到主存储器。任何形式的信息都可存储到文件中，包括程序、信函以及存货单等。

计算机允许连接几种不同的辅助存储设备。最常见的有硬盘、软盘、CD、DVD 和可移动闪存驱动器。计算机使用的 CD 和音乐 CD 基本相同，而 DVD 和视频 DVD 基本相同。计算机使用的 CD 和 DVD 可以是只读的，只能读取上面的数据，不可更改；也可以是可读/可写的，计算机可以更改上面的数据。**硬盘**通常固定在计算机内，不能随便取出。相反，**软盘**和 CD 很容易从驱动器中取出，并拿到另一台计算机上使用。软盘和 CD 的优点在于便宜和易于携带，但硬盘能存储更多的数据，而且速度更快。如今，**闪存驱动器**已在很大程度上取代了软盘，使用名为闪存的存储介质来存储数据。和主存储器不同，闪存驱动器即使没有供电，上面存储的数据也不会丢失。虽然还有其他形式的辅助存储设备，但这些形式是最常见的。

主存储器除了简称为**内存**，还可简称为 **RAM**，也就是**随机存取存储器**(Random Access Memory)。之所以是“随机存取”，是因为计算机能直接访问任意内存位置。辅助存储设备则通常要求顺序访问。换言之，计算机必须检索全部(或至少大量)存储位置，直至找到需要的数据项。

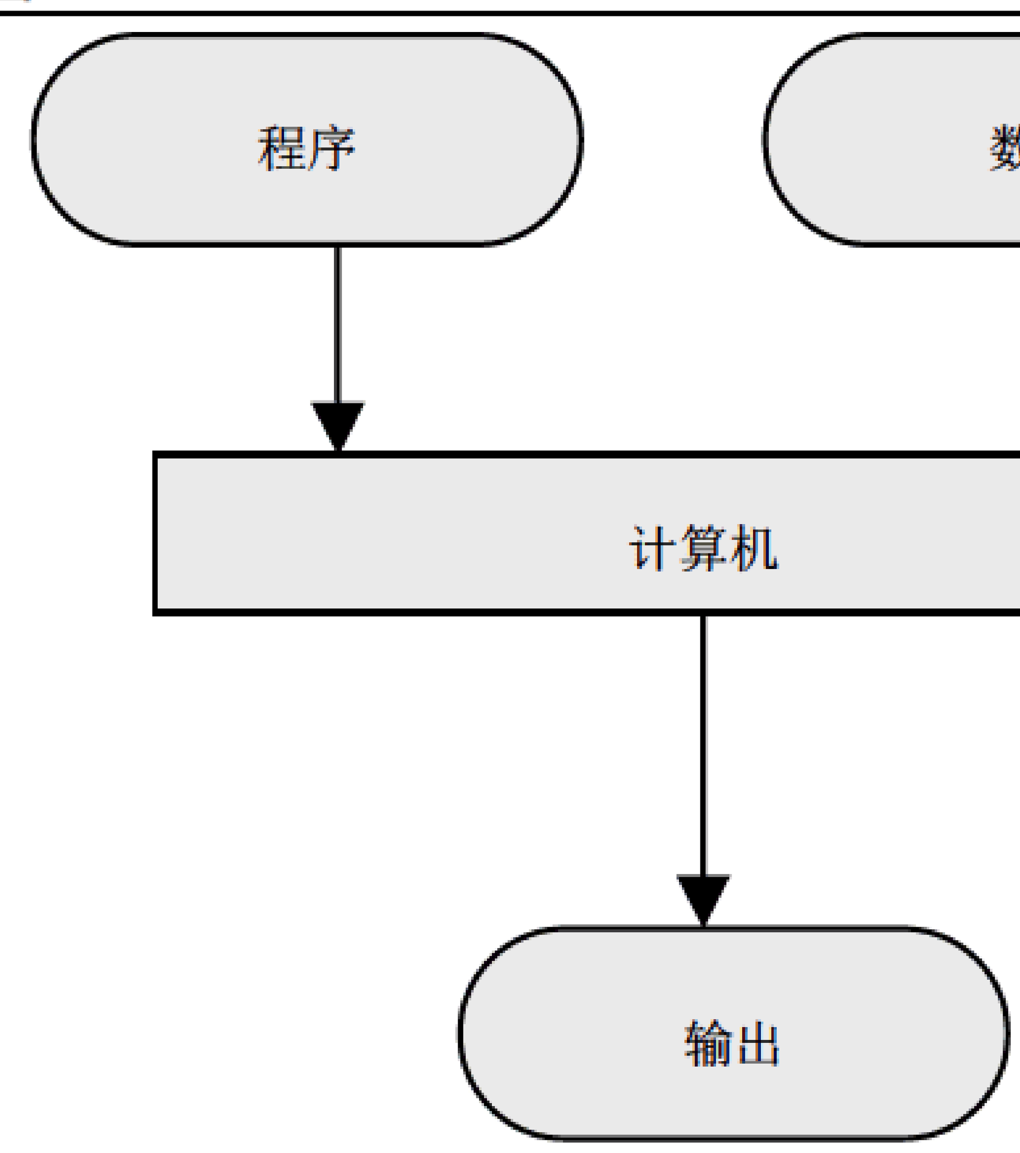
处理器(也称为中央处理单元，或者 CPU)是计算机的“大脑”。在广告中，厂家会说计算机用的是什么芯片。**芯片**指的就是处理器。处理器按程序指令操作，执行程序指定的计算。然而，处理器只是一个非常简单的“大脑”。它唯一能做的就是按照程序员提供的一系列简单指令进行操作。一般的处理器指令是：“将这个 0,1 序列解释成数字，将内存位置 37 的数字加到内存位置 59 的数字上，将结果放到位置 43”。或者“读取输入的一个字母，用 0,1 序列对其进行编码，将这个编码放到内存位置 1298”。处理器可执行加、减、乘、除，并可将数据从一个内存位置移动到另一个。它能将 0,1 序列解释成字母，并将字母发送到输出设备。处理器还具有重新排列指令顺序的基本功能。不同的计算机可能使用不同的处理器指令集。现代计算机的处理器通常都支持几百条指令。然而，如上文所述，每条指令执行的任务都是非常简单的。

软件

人一般不直接和计算机通信，而是通过操作系统和它交互。**操作系统**为计算机必须完成的不同任务分配计算机资源。操作系统实际是一个程序，或者是多个相互协作的程序，但更好的办法是把它想象成你的管家，负责管理家中的其他所有佣人，把你的要求传达给他们。如果希望运行一个程序，就要把包含这个程序的文件的名称告诉操作系统，操作系统会帮你运行它。如果想编辑一个文件，告诉操作系统文件名是什么，它会启动一个编辑器并加载那个文件。对于大多数用户，操作系统就是计算机。没有操作系统，大多数用户根本无法操作计算机。常用的操作系统包括 UNIX，DOS，Linux，Windows，Mac OS，iOS 和 Android 等。

程序是计算机需要遵照执行的一系列指令。如图 1.3 所示，计算机的输入由两部分组成：程序和数据。计算机按照程序中的指令操作，在这个过程中，会执行一些具体的处理。从概念上说，**数据**是向一个程序提供的输入。例如，如果程序要将两个数字加到一起，这两个数字就是数据。换言之，数据是程序的输入，而程序和数据共同构成了一台计算机的输入(一般通过操作系统)。任何时候只要向计算机提供了一个它必须遵照执行的程序，并为程序提供了一些数据，就称为要对那些数据**运行程序**，而计算机要对那些数据**执行程序**。“数据”一词还有更常规的含义。从广义上说，它意味着计算机可用的任何信息。狭义和广义的“数据”我们平时都在使用。

图 1.3　程序运行简单示意图



高级语言

可以使用许多语言编写程序。本书将讨论 C++编程语言，并用它编写程序。和其他大多数语言一样，C++也是一种高级语言。其他高级语言还有 C，C#，Java，Python，PHP，Pascal，Visual Basic，FORTRAN，COBOL，Lisp，Scheme 和 Ada 等等。**高级语言**在许多方面都类似于人类使用的语言，其设计宗旨是方便人们编写和阅读程序。高级语言包含的指令比 CPU 能够执行的简单指令要复杂得多。

计算机能理解的语言称为**低级语言**。在不同类型的计算机上，低级语言的细节也是不同的。一个典型的低级语言指令可能如下：

```
ADD X Y Z
```

它的意思是“将内存位置 X 的数字加到内存位置 Y 的数字上，再将结果放到内存位置 Z 处”。上述简单指令是用**汇编语言**写成的。虽然汇编语言已非常接近计算机能直接理解的语言，但仍要经历一次简单转换，才能真正被计算机理解。计算机要想遵照汇编语言指令行事，所有单词都必须转换成 0,1 序列。例如，单词 ADD 可能转换成 0110，X 可能转换成 1001，Y 转换成 1010，而 Z 转换成 1011。所以，执行上述汇编语言指令时，计算机实际执行的是下面的指令：

```
0110 1001 1010 1011
```

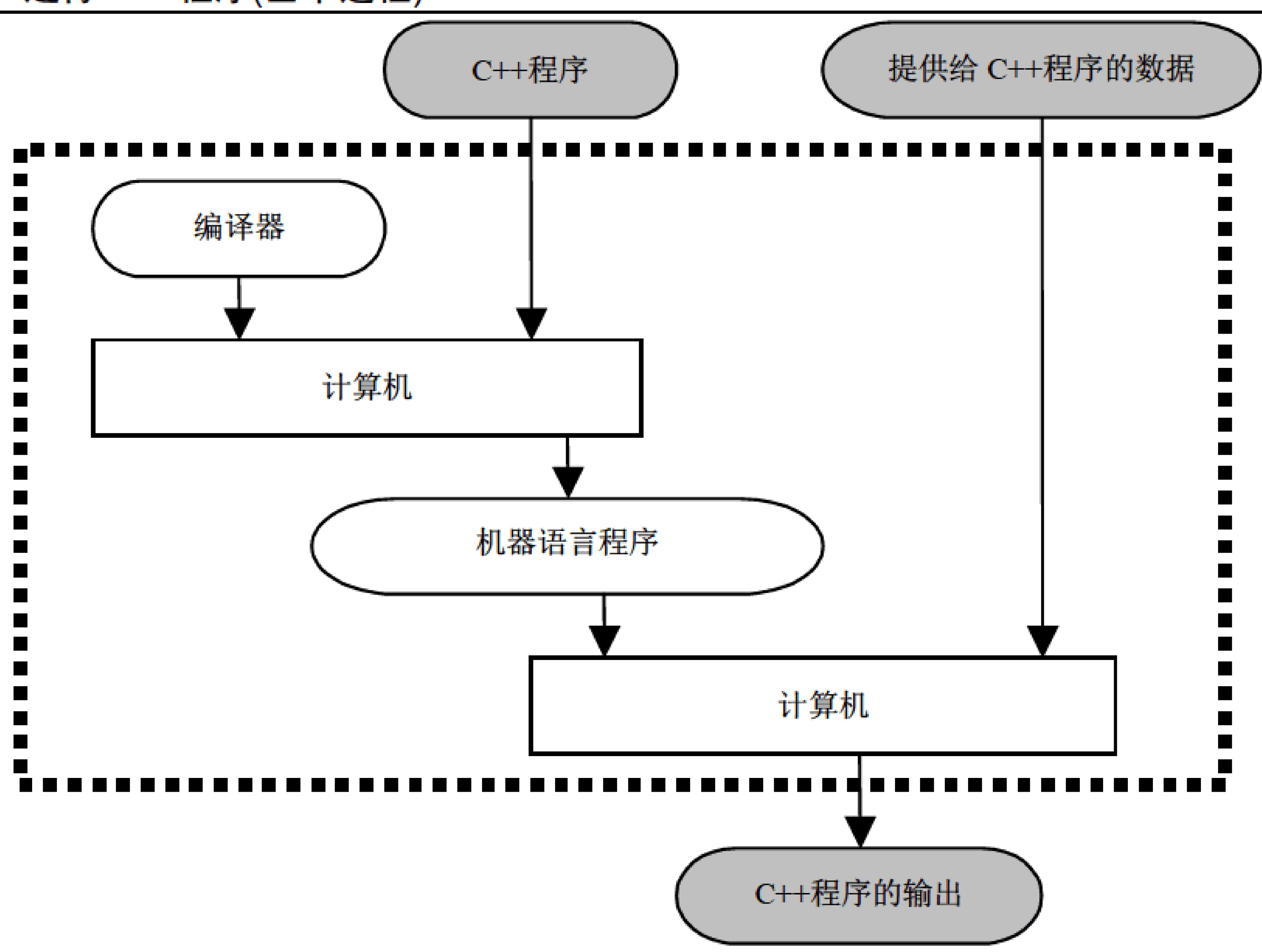
不同机器使用的汇编语言指令以及它们转换成 0,1 序列的方式是不同的。

这种 0,1 形式的程序是用**机器语言**写的，那才是计算机真正理解的语言。汇编语言和机器语言差别不大，而且这种差别对我们来说并不重要。重要的是机器语言和高级语言(比如 C++)的区别：用高级语言写的所有程序都必须翻译成机器语言，以便计算机理解。

编译器

将 C++等高级语言翻译成机器语言的程序称为**编译器**。编译器是一种特殊程序，它的输入(或数据)是一个程序，输出的又是一个程序。为避免混淆，一般将输入程序称为**源程序**或**源代码**，编译器输出的程序则称为**目标程序**或**目标码**。**码**或**代码**一词常用于表示程序或程序的一部分。说到目标程序时，通常会使用目标码一词。现在，假定你希望运行自己写的一个 C++程序。为了让计算机遵循 C++指令，要采取以下步骤。首先，运行编译器，将 C++程序作为数据提供给编译器。注意，在这种情况下，C++程序不被视为一系列要执行的指令。对于编译器来说，C++程序不过是一系列字符串。输出的仍是一系列字符串，即 C++程序的机器语言版本。接着，要运行这个机器语言版本，向它提供平常我们以为是提供给 C++程序的数据。如果有两台计算机，就更容易理解这一基本过程，如图 1.4 所示。但事实上，这个过程是在同一台计算机上完成的，只不过用了两次计算机。

图 1.4　编译和运行 C++程序(基本过程)

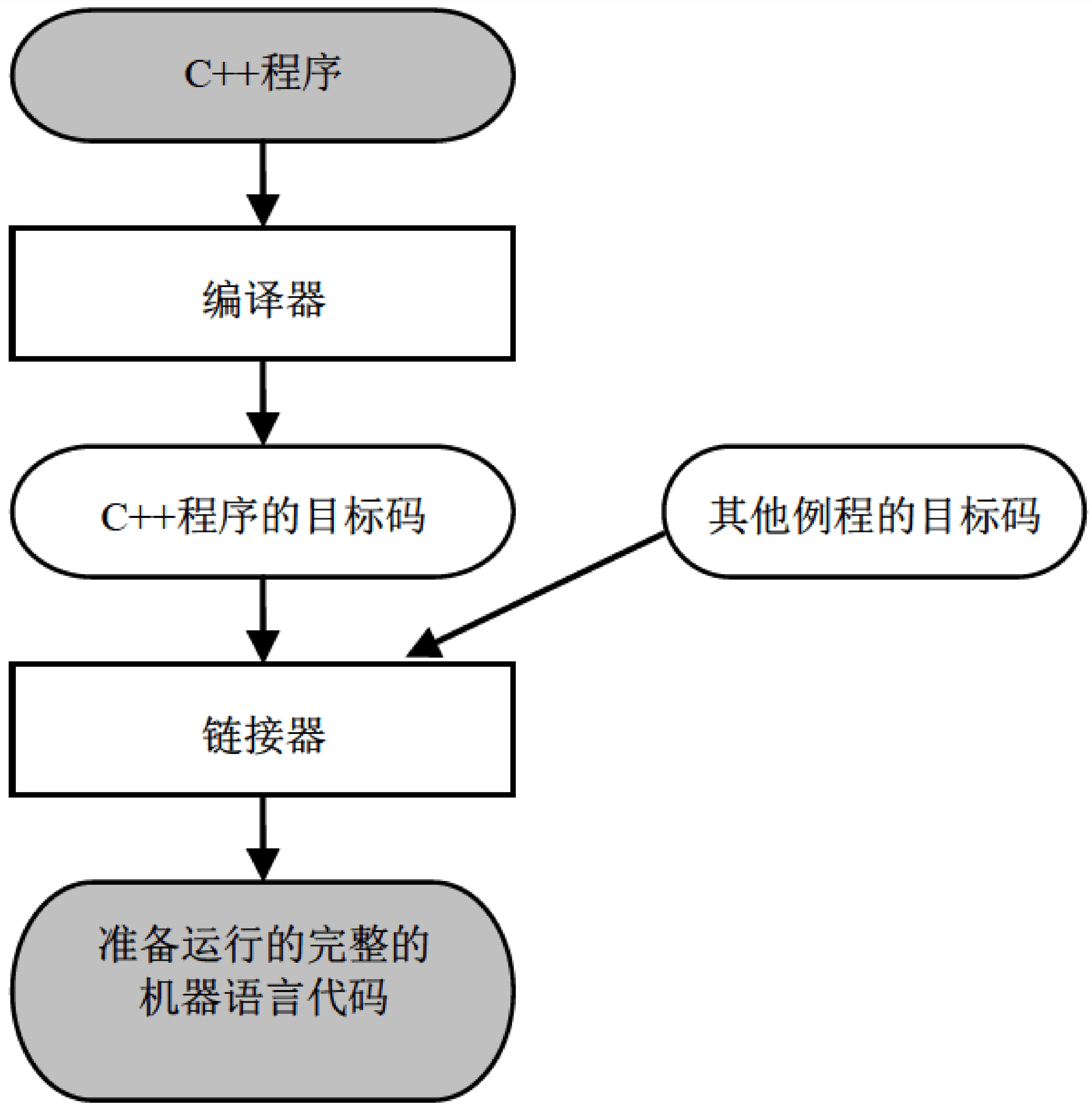


> **编　译　器**
>
> **编译器**是一种特殊的程序，它能将高级语言程序(比如 C++程序)翻译成机器语言程序，使计算机能直接理解并执行。

翻译和运行 C++程序的完整过程比图 1.4 的“基本过程”稍微复杂一些。任何 C++程序都会用到一些已编制好的操作(比如输入和输出例程)。这些操作已进行了编译，并生成了相应的目标码。它们等待着与你的程序的目标码合并，以生成一个完整的能够在计算机

中运行的机器语言程序。在这个过程中，要由另一个名为**链接器**的程序将事先准备好的目标码与基于你的 C++程序而生成的目标码合并。图 1.5 展示了编译器与链接器的交互。但在目前的大多数系统中，这个链接过程都是自动完成的。所以，一般不用关心链接问题。

图 1.5 运行 C++程序之前的准备工作



> **链 接**
>
> C++程序的目标码必须与程序用到的例程(比如输入和输出例程)的目标码合并。合并目标码的过程称为**链接**，由名为**链接器**的程序完成。简单程序的链接过程可能自动完成。

√ 自测题

1. 计算机的 5 个主要部件是什么？
2. 一个对两个数字进行相加的程序，它的数据是什么？
3. 一个为学生分配字母成绩的程序，它的数据是什么？
4. 机器语言程序和高级语言程序的区别是什么？
5. 编译器有什么作用？
6. 什么是源程序？什么是目标程序？
7. 什么是操作系统？
8. 操作系统的作用是什么？
9. 你上这门课时，计算机运行的是什么操作系统？
10. 什么是链接？
11. 你用的编译器是否具有自动链接功能？

历史回顾

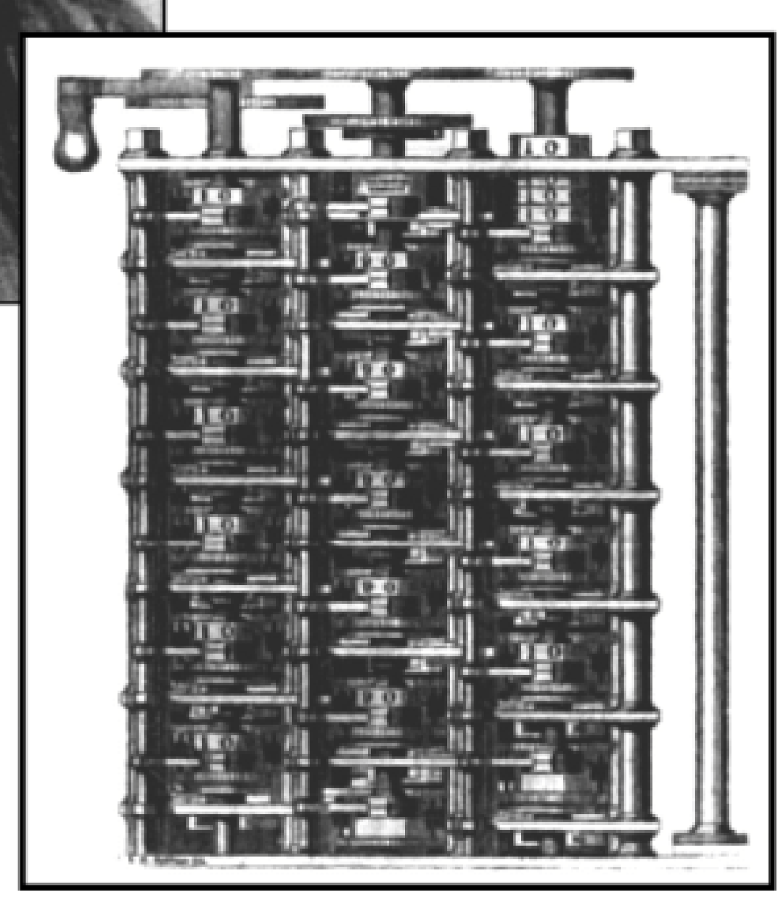
第一台真正可编程的计算机是由英国数学家和物理学家查尔斯·巴贝奇设计的。巴贝奇在 1822 年前的某个时间开始这个项目，并将自己的余生都奉献给了它。虽然这台机器始终没有完成，但它的设计思想是计算机历史上的一个里程碑。我们对于查尔斯·巴贝奇和他的计算机设计的认识，主要来自他的同事艾达·奥古斯塔的著作。艾达·奥古斯塔是诗人拜伦之女，后来成为伯爵夫人。许多人都认为艾达·奥古斯塔是有史以来的第一位计算机程序员。她的一些观点(在下一节开头引用)至今仍然适用于计算机的问题求解过程。计算机不具备魔法，不能(至少目前不能)为我们遇到的所有问题都自动给出妥善的解决方案。计算机只能做程序员要它做的事情。虽然解决方案最终由计算机执行，但方案本身由程序员自己制定。所以在讨论计算机编程时，首先要讨论程序员如何制定解决方案。



▲ 艾达•奥古斯塔伯爵夫人，第一位计算机程序员



▲ 查尔斯•巴贝奇



◀ 巴贝奇的计算机模型

1.2　编程和问题求解

分析机没有自主性。它能做我们操控它做的事情。它能跟着分析走；但无法预见到任何分析关系或定理。它的职责就是帮助我们做我们已知如何做的事情。

——*艾达·奥古斯塔，拉夫拉斯伯爵夫人(1815—1852)*

本节介绍设计和编写程序的常规原则。这些原则并非 C++特有，而是适合任何编程语言。

算法

开始学习自己的第一种编程语言时，很容易得出这样的观点：用计算机解决问题时，最困难的是如何将自己的思想转换成计算机能理解的语言。但实际并非如此。事实上，用计算机解决问题时，最困难的是找出解决问题的方案。只要有了解决方案，就能像例行公事那样将方案转换成需要的语言(无论 C++还是其他编程语言)。所以，有必要暂时忽略编程语言，将重点放在解决问题的步骤上，用通俗易懂的话将各个步骤写下来，就像是要向人而不是计算机发指令。以这种方式表示的指令序列通常称为算法。

用于解决问题的一系列准确的指令称为**算法**，通常也可以称为方法、指示、过程和例程等。指令可用编程语言或自然语言表示。我们的算法用中文和 C++编程语言来写。计算机程序就是用计算机能理解的语言来表示的算法。所以，“算法”一词比“程序”更常规。然而，我们在说一个指令序列是算法时，通常是说这些指令要用中文或英文描述。相反，如果是用编程语言描述的，就应该使用更具体的“程序”一词。下面这个例子有助于澄清这一概念。

图 1.6 用中文描述了一个算法，它判断指定的名字在名单中出现的次数。假定这个名单包含了上赛季的全部获胜球队，而且你想查找自己喜爱的球队的名字，这个算法就能判断你的球队赢了多少场。算法很短，但很典型。

图 1.6　示例算法

判断一个名字在名单中出现的次数

1. 获取名单。
2. 获取要检查的名字。
3. 将一个计数器设为零。
4. 为名单中的每个名字都采取以下操作：
   将名单中的当前名字与要检查的名字进行比较；
   如果两个名字相同，就在计数器上加 1。
5. 宣布答案是计数器当前的数字。

在上述算法中，编号为 1～5 的指令按照它们列出的顺序逐步执行。除非专门说明，否则总是假定一个算法的指令按照它们给定(写下)的顺序来执行。但是，大多数算法都要求能够更改执行顺序，例如重复执行其中的部分指令(比如上述算法中的指令 4)。

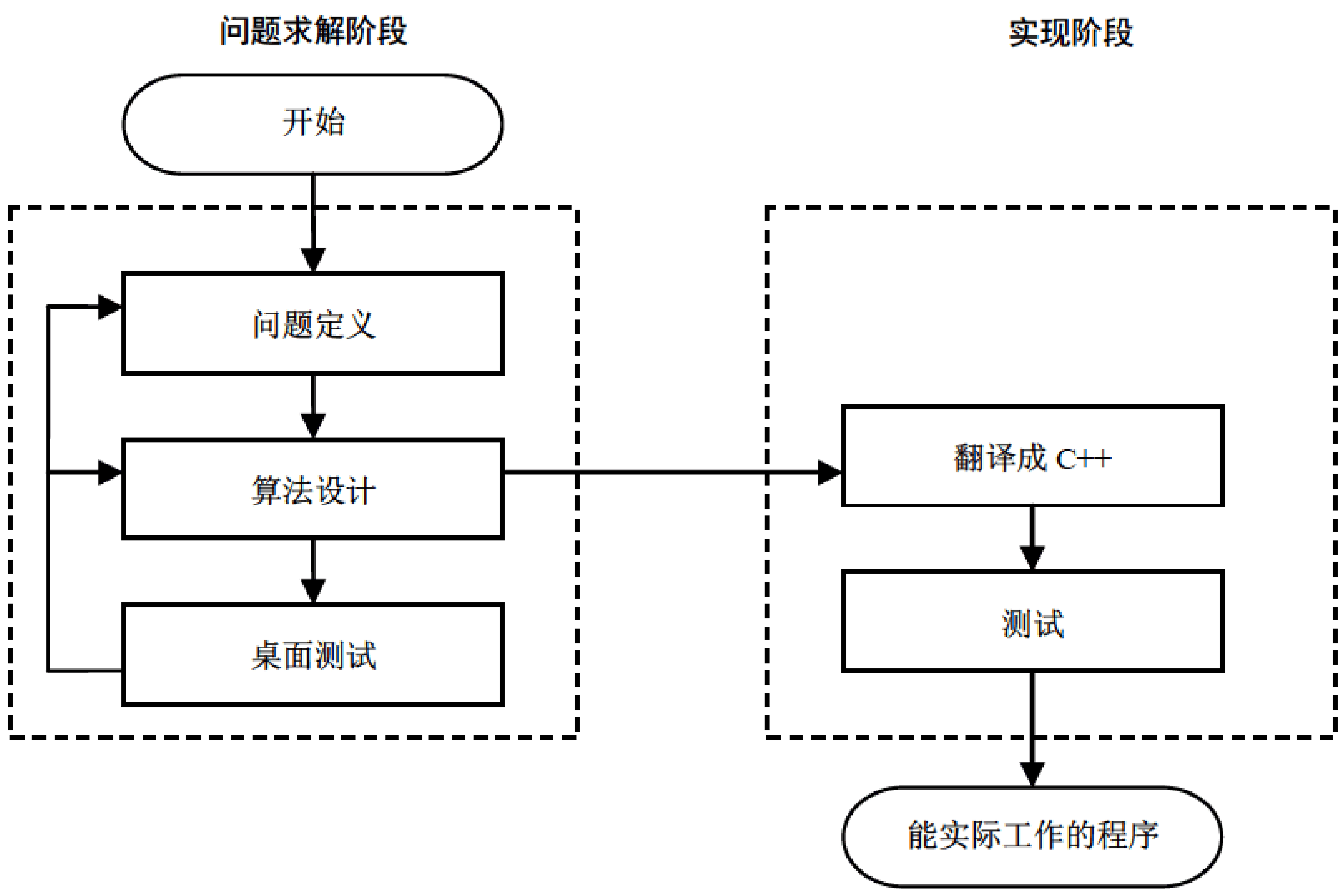
“算法”一词有悠久的历史。它来源于 19 世纪波斯数学家和天文学家阿尔 • 花拉子米(al-Khowarizmi)。他写了一本著名的关于数字和方程的教科书《重聚和求简规则》。英语的 algebra(代数)一词就来源于阿拉伯单词 al-jabr，这个词出现在书的标题中，通常直译成“重聚”(reuniting)或 restoring(还原)。历史上，代数(algebra)和算法(algorithm)这两个词的关系远比今天更为密切。事实上，“算法”近代以前一般专指求解数值方程的代数规则。如今，“算法”一词有了更广泛的含义，可用它表示对符号与数值数据进行处理的各种指令。指令序列能否成为合格的算法，要取决于指令的本质，而不是取决于指令所操纵的对象。要成为合格的算法，指令序列必须完整且无歧义地指定要执行的各个步骤，以及执行这些步骤的顺序。人或机器必须完全按算法行事，不多也不少。

> **算　　法**
>
> **算法**是能使问题得以解决的一系列准确的指令。

程序设计

设计程序通常是一个困难的任务。没有一套完整的规则，也没有算法告诉你怎样写程序。程序设计是创新过程。但是，仍有一个过程纲要可循。图 1.7 展示了这个纲要。如图所示，整个程序设计过程可划分为两阶段，即问题求解阶段和实现阶段。**问题求解阶段**的结果是用于解决当前问题的算法，通常用中文或英文描述。为了用 C++等编程语言生成程序，算法需要翻译为编程语言。根据算法生成最终程序就是**实现阶段**的任务。

图 1.7　程序设计过程



第一步是确定任务(想让程序做的事情)得到了完整、准确的定义。不要轻视这个步骤的重要性。如果自己都不确定程序应该产生什么输出，看到真正的输出时，可能会大吃一惊。一定要确定程序应该获得什么输入，而且要知道提供了特定输入之后，程序的输出结果以及输出结果的格式。例如，对于一个银行会计程序，不仅要知道利率，还要知道利率是否需要每年、每月或者每日进行复利计算。对于一个自动写诗程序，需要确定输出的是自由体、抑扬格还是其他诗体。

许多涉足编程领域的新手都不理解为什么要先设计好算法，再用编程语言(比如 C++)写真正的程序。他们可能完全省略“问题求解阶段”，或者只做其中的“问题定义”部分。这表面上是合理的。为什么不能“直奔目标”，节省一些时间呢？答案很简单：“这根本节省不了时间！”经验证明，将程序设计过程分为两阶段，能更快生成一个能正确工作的程序。两阶段的程序设计过程简化了算法设计，因为在进行算法设计时，不需要关心编程语言(比如 C++)的详细规则。结果是算法设计过程的复杂程度大大降低，出错概率也大幅降低。即使很小的程序，这样做也有好处。你是希望进行半天有条不紊的工作，就设计出

一个完善的程序？还是想进行几天令人抓狂的工作，对一个不好理解的程序进行排错呢？

实现阶段并不轻松。一些细节需要关注，其中一些可能很容易被忽略。但是，相较于许多人最初的想象，这个步骤还是要简单得多。一旦熟悉了 C++或者其他任何编程语言，将算法从自然语言(比如中文)翻译成编程语言，就像例行公事那样简单。

如图 1.7 所示，两个阶段都需要测试。写程序前要测试算法，如果发现算法存在不足，必须重新设计算法。对于一个不大的程序，为了进行桌面测试，只需在心里面跟着算法走几遍，自己执行算法的各个步骤。如果算法较复杂，可能需要一支笔和一张纸。为了测试 C++程序，只需编译并运行它，并提供一些样本输入数据。编译器会为一些特定类型的错误给出准确的错误消息。但对于其他类型的错误，则必须对输出进行检查，自己验算是否出错。

图 1.7 展示的是理想情况下的程序设计过程。应该记住这个基本流程，但在实际应用中，设计过程也许要复杂一些。实际编程时，错误和缺陷会不定期显现出来，而你可能必须退回去，重做以前的步骤。例如，通过测试算法，你可能发现问题定义还不完善。在这种情况下，就必须退回去，重新定义问题。另一些时候，问题定义或算法中的缺陷要等到测试程序时才会显现。在这种情况下，就必须回头修改问题定义或算法，然后重新执行它们之后的所有步骤。

面向对象编程

在 1.2.2 节概括的程序设计过程中，将程序描述成对数据进行操纵的算法(指令序列)。这虽然正确，但并非总是最有效的。现代程序往往采用名为**面向对象编程**(Object Oriented Programming，OOP)的方法进行设计。在 OOP 中，程序被视为一系列交互对象的集合。以模拟程序为例，可以很容易地理解这种方法。例如，假定程序要模拟公路立交桥，对象就可能包括汽车和车道。每个对象都用自己的算法描述它在不同情况下的行为。进行 OOP 编程时，需要设计对象及其所用的算法。用 OOP 的框架进行编程，图 1.7 的“算法设计”应替换成“设计对象及其算法”。

OOP 的主要特点是封装、继承和多态性。**封装**通常被描述成一种信息隐藏(或者抽象)形式。虽然这样说没有问题，但这种说法更容易理解：封装是“对象描述”的简化形式。**继承**涉及编写可重用的代码。而**多态性**是指一个名称在继承的上下文中具有多种含义。虽然简单描述了 OOP，但必须承认，对于从未听说过 OOP 的读者，这些描述的意义不大。但不用担心，所有这些术语都将在本书后面进行详细解释。C++提供了**类**来顺应 OOP 风格的编程。类是整合了数据与算法的一种数据类型。

软件生命周期

大型软件系统(比如编译器和操作系统)的设计者将软件开发过程划分为 6 个阶段，这 6 个阶段统称为**软件生命周期**，具体如下：

1. 任务分析和规范化(问题定义)；
2. 软件设计(对象和算法设计)；
3. 实现(编码)；

4. 测试；
5. 维护和系统演化；
6. 废弃。

讨论程序设计时没有提及最后两个阶段，因为它们是在程序完成并投入使用之后才开始的。但是，心中要一直记住它们。只有将程序设计得易于理解和修改，才好对其进行增强或修改。如何设计程序，使其易于修改，这将是一个重要主题。掌握了更多的背景知识和编程技术之后，就会详细讨论到它。“废弃”的含义不言而喻，只是许多人不愿接受而已。假如程序不能像设计的那样工作，对其进行修改也得不偿失，就应该把它废弃，用一个全新程序取代它。另外，假如老程序不能胜任新工作，而且不便在它的基础上重构，也应废弃不用。

√ 自测题

12. 食谱和算法很相像。但是，食谱允许的一些指令在算法中不允许。对于以下食谱，哪些指令在算法中是允许的？
    - 在搅拌碗里加两汤匙糖
    - 在搅拌碗里加 1 个鸡蛋
    - 在搅拌碗里加 1 杯牛奶
    - 加 1 盎司(1 oz ≈ 28.35 g)朗姆酒(喝酒不开车)
    - 根据口味添加香草精
    - 搅拌均匀
    - 倒入一个漂亮的玻璃杯
    - 撒上肉豆蔻
13. 创建程序时，第一步是什么？
14. 程序设计过程可划分为哪两个主要阶段？
15. 为什么不能轻视问题求解阶段？请加以解释。

1.3　C++入门

语言只是科学的工具……

——塞缪尔·约翰逊[①](1709—1784)

本节介绍 C++编程语言，它是本书使用的编程语言。

C++语言的起源

对于 C++语言，人们首先关注的是它不同寻常的名字。你也许会问，还有 C 语言吗？有 C−或 C−−语言吗？有 A 和 B 吗？大多数问题的答案都是“没有”。但是，提出这些问题的出发点是好的。确实有一种 B 语言，但它不是从一种名为 A 的语言派生的，而是派生自 BCPL 语言。C 语言是从 B 语言派生的，C++是从 C 语言派生的。那么，C++的两个加

① 英国词典学家，编撰有历史上第一本广泛使用的、影响巨大的英文字典。——译注

号是怎么来的？如同在第 2 章要学到的那样，++是 C 和 C++语言支持的一种操作，所以++是一个不错的双关语。我们对 BCPL 和 B 都不感兴趣，它们是 C 语言的早期版本。我们将从介绍 C 语言开始 C++之旅。

C 语言问世于 20 世纪 70 年代，发明人是 AT&T 贝尔实验室的丹尼斯 • 里奇(Dennis Ritchie)。它最初用于编写和维护 UNIX 操作系统(在此之前，UNIX 系统程序要么只能用汇编语言来写，要么只能用 B 语言来写。B 语言由肯 • 汤普生(Ken Thompson)发明，他是 UNIX 的缔造者)。C 是一种通用语言，可用来写任何类型的程序，但 UNIX 操作系统直接促成了它的成功与普及。要维护 UNIX 操作系统，就需要使用 C。C 和 UNIX 的配合是如此天衣无缝，以至于不久以后，不仅系统程序，就连 UNIX 下运行的几乎所有商业程序都开始用 C 语言来编写。随着 C 越来越流行，逐渐出现了为其他流行操作系统编写的 C 语言版本，C 的应用开始不受 UNIX 机器的局限。不过，虽然 C 非常流行，但并不是完美无缺的。

C 语言的特殊性在于，它虽然是一种高级语言，但又具有低级语言的大量特点。C 其实介于一种非常高级的语言和一种低级语言之间，优点和缺点都很突出。类似于(低级)汇编语言，C 语言程序可直接操纵计算机的内存。另一方面，C 又具有高级语言的许多特点，所以比汇编语言更容易理解和编写，这使 C 成为编写系统程序的理想选择。但在编写其他程序(有时甚至包括一些系统程序)时，C 不像其他高级语言那样容易理解。另外，它不像其他高级语言那样具有完善的自动检查功能。

为了解决上述问题以及 C 的另一些缺陷，AT&T 贝尔实验室的本贾尼 • 斯特劳斯特卢普(Bjarne Stroustrup)在 20 世纪 80 年代初发明了 C++。他把 C++设计成一种更好的 C。C 的很大一部分都成为C++的一个子集，所以大多数C程序其实也是C++程序(反之则不成立，许多 C++程序都绝非 C 程序)。和 C 不同，C++具备了“面向对象编程”(OOP)的能力。前面讲过，OOP 是一种非常强大的编程技术。

一个 C++示范程序

图 1.8 是一个简单的 C++程序，用户运行这个程序并和它交互，就能生成屏幕显示。运行程序的人称为**用户**。程序运行时的输出在“示范对话”中展示。用户输入的文本以粗体显示，以便和程序生成的文本区分。但在实际屏幕显示中，两种文本看起来一样。1～22 行是程序源代码。行号仅供参考。输入程序时不要输入行号。写程序的人叫**程序员**。虽然用户和程序员是不同角色，但用户和程序员可能是、也可能不是同一个人。例如，如果你写程序并运行它，你就既是程序员，也是用户。对于专业编写的程序，程序员和用户通常是不同的人。

第 2 章将解释为了写图 1.8 的程序需要掌握哪些 C++知识。但为了使你对 C++程序有一个初步印象，下面简单解释一下这个特定的程序是如何工作的。不用担心自己不清楚一些细节，本节只是让你对 C++程序有一个感性认识。

图 1.8　一个 C++示范程序

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5      int numberOfPods, peasPerPod, totalPeas;
```

```
6     cout << "Press return after entering a number.\n";
7     cout << "Enter the number of pods:\n";
8     cin >> numberOfPods;
9     cout << "Enter the number of peas in a pod:\n";
10    cin >> peasPerPod;
11    totalPeas = numberOfPods * peasPerPod;
12    cout << "If you have ";
13    cout << numberOfPods;
14    cout << " pea pods\n";
15    cout << "and ";
16    cout << peasPerPod;
17    cout << " peas in each pod, then\n";
18    cout << "you have ";
19    cout << totalPeas;
20    cout << " peas in all the pods.\n";
21    return 0;
22 }
```

示范对话

```
Press return after entering a number.
Enter the number of pods:
10
Enter the number of peas in a pod:
9
If you have 10 pea pods
and 9 peas in each pod, then
you have 90 peas in all the pods.
```

示范程序的开始与结束处包含目前不用关心的一些细节。程序从这几行代码开始：

```
#include <iostream>
using namespace std;

int main()
{
```

就目前来说，可想象这些行以一种非常复杂的方式表示“程序从此开始”。

程序以下面这两行结束：

```
    return 0;
}
```

对于简单的程序，这两行表示“程序在此结束”。

开始与结束部分之间的代码是程序的核心。我们将简单解释一下这些代码，首先是下面这一行：

```
int numberOfPods, peasPerPod, totalPeas;
```

这一行是**变量声明**，告诉计算机 numberOfPods(豆荚数)，peasPerPod(每豆荚的豆子数)和 totalPeas(豆子总数)将作为 3 个变量名使用。第 2 章将更详细地介绍变量，但它们在这个程序中的用法很容易理解。程序用**变量**命名数字。打头的 int 是单词 integer(整数)的缩写，告诉计算机这些变量命名的数字是整数。**整数**是包括 0 在内的所有能被 1 整除的数，如 1, 2, -1, -7, 0, 205, -103，等等。

剩余的行都是告诉计算机做某事的指令。这些指令称为**语句**或**可执行语句**。在这个程序中，每个语句刚好占一行。虽然并非一定要这样，但对于简单程序来说，语句通常都是一行一行地写。

大多数语句以单词 cin 或 cout 开头。这些是输入语句和输出语句。cin 读作“see-in”，用于输入。以 cin 开头的语句告诉计算机当用户从键盘输入信息时应该做什么。cout 读作“see-out”，用于输出，也就是将信息从程序发送到终端屏幕。字母 c 来源于语言的名称(C++)。箭头 (<<或>>) 指示数据移动方向。<<和>>分别称为“插入”和“提取”，或者“送入”和“获取”。例如下面这一行：

```
cout << "Press return after entering a number.\n";
```

它应该这样读：将 "Press...number.\n" 送入 cout，或简单地读作：输出 "Press...number.\n"。将 cout 想象成屏幕(输出设备)的名称，就很容易理解箭头的含义，它告诉计算机将引号中的字符串发送给箭头指向的目标，也就是屏幕。如示范对话所示，这导致引号中的文本输出到屏幕。\n 是换行符，告诉计算机在输出完文本后换行。下一行代码也以 cout 开头，导致程序将以下文本输出到屏幕：

```
Enter the number of pods:
```

下一行以单词 cin 开头，所以是输入语句。下面这行代码：

```
cin >> numberOfPods;
```

可以这样读：从 cin 获取 numberOfPods，或简单地读作：输入 numberOfPods。

将 cin 视为键盘(输入设备)，很容易理解箭头的含义，它告诉计算机将来自键盘的输入发送给箭头指向的目标(变量 numberOfPods)。再来看示范对话，下一行显示了一个粗体的 10。我们用粗体表示用户通过键盘输入的内容。输入数字 10，屏幕上就会出现 10。再按 Enter 键，10 就会输入程序。以 cin 开头的语句告诉计算机将那个输入值(10)发送给变量 numberOfPods。所以在此之后，numberOfPods 就有了一个值 10。以后再在程序中看到 numberOfPods 时，就可以认为它代表数字 10。

再来看下面这两行代码：

```
cout << "Enter the number of peas in a pod:\n";
cin >> peasPerPod;
```

它们与前面两行代码非常相似。第一行代码将一条消息发送到屏幕上，要求用户输入一个数字。通过键盘输入数字并按 Enter 键，输入的数字就会成为变量 peasPerPod 的值。示范对话假设输入的是 9。输入 9 并按 Enter 键，变量 peasPerPod 的值就变成 9。

下一行代码(不计空行)将执行计算，这是这个简单的程序唯一执行的计算：

```
totalPeas = numberOfPods * peasPerPod;
```

星号(*)在 C++中表示乘法运算。所以上述语句会使 numberOfPods 和 peasPerPod 相乘。在这个例子中，10 乘以 9 等于 90。等号表示变量 totalPeas 应该等于结果值 90。这是等号的一种特殊用法，它在程序中的含义与在数学中不同。在 C++程序中，它向左侧的变量赋一个值(可以是新值)。在本例中，它使 90 成为 totalPeas 的值。

程序剩余的部分是形式基本相同的输出，随后三行(不计空行)代码是：

```
cout << "If you have ";
cout << numberOfPods;
cout << " pea pods\n";
```

这三行输出语句与前面以 cout 开头的语句基本相同。唯一要注意的就是第二行，它要求输出变量 numberOfPods。输出变量时，实际输出的是变量的值。所以这个语句会导致输出 10(记住，在程序的示范运行过程中，用户将变量 numberOfPods 设为 10)。所以，上述三行输出语句的结果是：

```
If you have 10 pea pods
```

注意，输出的内容都在同一行上。除非输出\n，否则不换行。

程序剩下的没有什么新东西，不需要赘述。如果理解了前面讨论的所有内容，剩下的语句应该能轻松理解。

陷阱：在\n 中错误地使用斜杠

在 cout 语句中使用\n 时，必须使用反斜杠，也就是\。如果错误地写成/n，而不是\n，编译器不会给出错误消息。程序仍会运行，但输出可能会和你想象的不同。■

编程提示：输入和输出语法

将 cin 看成是键盘或输入设备的名称，将 cout 看成是屏幕或输出设备的名称，就很容易记住箭头>>和<<的方向。它们指示的是数据移动方向。例如以下语句：

```
cin >> numberOfPods;
```

数据从键盘向变量 numberOfPods 移动，所以箭头要从 cin 指向变量。

再来看看输出语句：

```
cout << numberOfPods;
```

数据从变量 numberOfPods 向屏幕移动，所以箭头要从变量指向 cout。■

简单 C++程序的布局

图 1.9 展示了简单 C++程序的布局。从编译器的角度看，换行和间隔[①]不必非要像例子中那样。编译器接受任何合理的换行和缩进风格。事实上，编译器甚至能接受许多不合理的换行和缩进风格。然而，程序的布局应该总是确保它的可读性。起始花括号{和结束花括号}单独占一行，会使这些标点符号更醒目。缩进每行语句并使其单独占一行，可以更容易地理解程序指令。以后的一些语句会比较长，以至于一行无法完全显示，这时就要稍微修改一下这种缩进和换行风格。你应该遵照本书范例的风格；在课堂上使用本书，请遵照教师指定的风格。

图 1.9　一个简单 C++程序的布局

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      变量声明
```

① 包括行间和词间的间隔，包括空格、制表符、空行等。——译注

```
7
8       语句1
9       语句2
10      …
11      语句n
12
13      return 0;
14  }
```

在图 1.8 中，变量声明是以单词 int 开头的那一行。正如第 2 章要说明的那样，不一定要将变量声明统统放在程序开头，但那是一个很好的默认位置。除非有特别的理由放到其他地方，否则应该像图 1.9 和图 1.8 那样放在程序开头。**语句**是计算机要遵照执行的指令。在图 1.8 中，语句是以 cout 或 cin 开头的那些行，以及以 totalPeas 和一个等号开头的那一行。语句通常也称为**可执行语句**，两种说法可以混用。注意，每个语句都以分号结尾。语句中的分号就像日常用语中的句号，用于标记一个句子的结尾。

就目前来说，可以认为前几行是在以一种奇怪的方式说“程序从此开始”，但是可以稍微详细地解释一下它们。第一行代码如下：

```
#include <iostream>
```

它称为 include **预编译指令**，告诉编译器在哪里寻找与程序中使用的特定项目有关的信息。在本例中，iostream 是一个库的名称，其中包含了输入和输出例程的定义。iostream 本质上是一个文件，其中含有与这个库有关的基本信息。本章前面讨论的链接器会将 iostream 库中的目标码与程序的目标码合并起来。对于 iostream 库，这个过程在你的系统上可能是自动完成的。你迟早都会用到其他库。使用时，必须在程序开头的预编译指令中指定它们的名称。除此之外，在使用一些库时，要做的可能不只是在程序中放一条 include 预编译指令那样简单。但无论如何，在程序中使用任何一个库，都至少要为那个库添加一条对应的 include 预编译指令。预编译指令总是以符号#开头。有的编译器要求#的前后都不能有空格。所以，最保险的做法是在一行的最开头写下#，而且不要在#和单词 include 之间插入空格。

第 2 行代码进一步解释了刚才的 include 预编译指令：

```
using namespace std;
```

这行代码指出：iostream 中定义的名称要以“标准方式”进行解析(std 就是标准的意思，它是 standard 的缩写)。本书稍后会更多地解释这一行。

第 3 行和第 4 行代码(如下所示)指出程序的 main 部分从这里开始：

```
int main()
{
```

正确说法是“main 函数”，而不是“main 部分”。但是，这个区分的意义要等到第 4 章才会真正明白。花括号{和}标记程序 main 部分的开始与结束。它们不需要单独占一行，但这样可以更方便地找到它们。倒数第二行代码如下：

```
return 0;
```

它表示“到这里就终止程序”。该行不一定位于程序末尾，但对于极其简单的程序，把它

放到其他地方没有意义。有些编译器允许省略该行，在发现没有更多可执行语句时会自动终止程序。但是，另一些编译器要求必须包括这一行，所以最好还是习惯于包括它。该行的正式名称是 `return` **语句**，可把它视为一个可执行语句，因为它也会要求计算机做某事。就目前而言，数字 `0` 对我们来说没有特别直观的含义，但它必须在那个地方。随着你更深入地学习 C++，就会真正明白它的含义。注意，虽然 `return` 语句表示要终止程序，但仍然要在程序的 `main` 部分末尾添加一个结束花括号}。

陷阱：在 include 的文件名前错误地添加一个空格

注意<iostream>的写法！在<和 iostream 这个文件名之间，以及在文件名末尾和>之间，一定不要包括任何多余的空格(参见图 1.9)。编译器在处理 include 预编译指令时，并没有你想象的那么聪明，它会真的搜索以一个空格开始或结束的文件名！这样的文件名是不存在的，所以会造成一个很难发现的错误。为了体会这一点，你可以试着在一个小程序中故意犯这个错误，编译它。记住编译器生成的消息，以便下一次看到同样的消息时，能迅速地判断为什么出错。■

编译和运行 C++程序

视频讲解：*Compiling and Running a C++ Program*

前面的小节解释了运行图 1.8 的 C++程序发生的事情。但是，程序具体在哪里？怎么使它运行起来呢？这是下面要讨论的主题。

C++程序用文本编辑器编写。这和写其他文档(比如学期报告、情书、购物单等)没有多大区别。程序和其他文档一样存储在文件中。有多种文本编辑器可供选择，每种文本编辑器的具体使用方式都有一定区别。所以，这里不便过多讲解文本编辑器的问题。如果有不明白的地方，请查阅编辑器的联机帮助。

编译和运行 C++程序的方式也取决于所用的系统，所以只能从常规角度讨论这个问题。你需要知道在自己的系统上编译、链接和运行 C++程序的命令是什么。命令可以在系统手册中找到，也可询问在你的系统上用过 C++的人。执行编译命令，会生成 C++程序翻译成机器语言的版本，该版本称为“目标码”。目标码必须与现有例程(比如输入和输出例程)的目标码链接(也就是合并)起来。链接过程可能是自动完成的，所以一般不必关心链接问题。但在某些系统上，仍然需要单独调用链接器。同样地，请查阅文档或询问其他人。最后是运行程序。具体如何做也要取决于系统，所以请查阅手册或询问别人。

陷阱：编译 C++11 程序

C++11(之前称为 C++0x)是 C++编程语言的最新标准，2011 年 8 月 12 日由国际标准化组织批准。C++14 于 2014 年 12 月 15 日发布，对 C++11 进行了少许扩展。本书不讨论这些扩展。C++11 编译器能编译和运行用老版本 C++写的程序。但 C++11 包含新的、和老的 C++编译器不兼容的语言特性。这意味着老的 C++编译器可能无法编译和运行 C++11 程序。

可能还要指定是否按照 C++11 标准来编译。例如，g++4.7 要求在命令行上添加编译器开关`-std=c++11`，否则编译器假定 C++程序是按旧标准写的。例如，以下命令行编译名

为 testing.cpp 的 C++11 程序。

```
g++ testing.cpp -std=c++11
```

查阅编译器文档了解是否需要特殊步骤编译 C++11 程序，并了解支持哪些 C++11 语言特性。 ■

编程提示：让程序运行起来

不同编译器和不同环境可能要求以略微不同的方式为 C++程序创建文件。请参考图 1.10 所示的程序。它和本书其他所有示例程序一样都可从网上下载(详情参见本书“前言”)。另外，也可自己录入程序并进行编译。如果出现错误消息，请检查程序，改正任何录入错误，再重新编译。如果程序成功通过编译，没有显示错误消息，就试着运行程序。[①]

图 1.10　测试你的 C++设置

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      cout << "Testing 1, 2, 3\n";
7      return 0;
8  }
9
```

如果不能编译和运行这个程序，请阅读前面的“编程提示：让程序运行起来”，它提供了在特定计算机配置中运行 C++程序的一些建议

示范对话

```
Testing 1, 2, 3
```

如果程序正常编译和运行，表明设置无误，不必修改本书的其他任何示例程序。如果程序无法编译，或者不能正常运行，请继续阅读本节的内容。我们将提供对 C++程序进行处理的一些建议。一旦让这个简单的程序正常运行起来了，就知道应该对其他 C++程序文件进行哪些小改动，使它们在自己的系统上正常运行。

如果程序表面上在运行，但看不到下面这一行输出：

```
Testing 1, 2, 3
```

表明程序极有可能已经产生了输出，只是在你看到之前便消失了。在程序尾部(`return 0;` 之前)添加下面这几行代码。这几行会使程序暂停，方便你看清输出：

```
char letter;
cout << "Enter a letter to end the program:\n";
cin >> letter;
```

① C++开发环境的选择非常灵活。使用 Visual Studio 有两个方案可供选择。如果希望所有工作都在 Visual Studio 中进行，请选择“文件”|“新建”|“项目”。语言选择“Visual C++”，选择“常规”类别的“空项目”模板，输入名称并指定存储位置后，在“解决方案资源管理器”中右击“源文件”并选择“添加”|“新建项”，在“代码”类别中选择“C++文件(.cpp)”，输入要创建的.cpp 文件的名称(自己添加.cpp 扩展名)。要执行程序，按 Ctrl+F5 或者选择“调试”|“开始执行(不调试)”。第二个方案是只用 Visual Studio 提供的 C++编译器 CL.exe，编辑器则选择一个自己喜欢的（例如 Notepad++）。为此，请执行 Visual Studio 程序组中的某个“命令提示”。在命令提示符窗口中，切换到.cpp 程序所在的目录，运行“cl *文件名*”来编译并生成程序。其他推荐的 IDE 还有 Dev C++，NetBeans，Eclipse 和 CodeLite。本书中文资源站（*transbot.ys168.com*）提供了这些工具的官网链接。——译注

现在整个花括号中的内容变成：

```
cout << "Testing 1, 2, 3\n";
char letter;
cout << "Enter a letter to end the program:\n";
cin >> letter;
return 0;
```

虽然目前可能还不理解新增的这些行，但到第 2 章结束时就能完全理解了。

如果程序完全无法编译和运行，试着修改下面这一行代码：

```
#include <iostream>
```

在 iostream 末尾添加一个.h，变成以下形式：

```
#include <iostream.h>
```

如果程序要求 iostream.h 而不是 iostream，则表明使用的是老版本 C++编译器，最好换用一个更新的编译器。

如果程序仍然无法编译和运行，试着删除下面这一行代码：

```
using namespace std;
```

如果程序还是无法编译和运行，请查阅你的这个版本的 C++的文档，了解是否需要为“控制台”(console)输入/输出增加更多的“预编译指令”(directive)。

如果一切努力均告失败，而且是在课堂上使用本书，请向教师寻求帮助。如果本书用于自学，或者用的不是学校的计算机，请查阅 C++编译器文档或者向计算机配置与你相似的朋友求助。事实上，为了解决这些问题，一般只需进行少量修改。一旦知道了怎样修改，就会发现原来一切都十分简单。 ■

√ 自测题

16. 如果在一个 C++程序中使用以下语句，会导致在屏幕上显示一些东西。具体会显示什么？

```
cout << "C++ is easy to understand.";
```

17. 以下语句(摘自图 1.8)中的符号\n 有什么含义？

```
cout << "Enter the number of peas in a pod:\n";
```

18. 以下语句有什么含义(摘自图 1.8)？

```
cin >> peasPerPod;
```

19. 以下语句有什么含义(摘自图 1.8)？

```
totalPeas = numberOfPods * peasPerPod;
```

20. 下面这个预编译指令有何含义？

```
#include <iostream>
```

21. 以下#include 预编译指令有错吗？错在哪里？

```
a. #include <iostream >
b. #include < iostream>
c. #include <iostream>
```

1.4　测试和调试

"你从三百六十五中去掉一，还余多少？"

"当然是三百六十四。"

矮胖子好像有点不相信，说："我倒要看看在纸上是怎么算的。"

——刘易斯·卡洛尔，《爱丽丝镜中奇遇记》

程序中的错误通常称为虫子(bug)，消除错误的过程称为**抓虫**、**除错**或**调试**(debug)。这个术语的来历非常有趣。在计算机发展早期，计算机硬件是相当容易损坏的，而且动辄占据整个屋子。霍普(Grace Murray Hopper，1906—1992))是"全球第一台大规模数字计算机的第三名程序员"。[①]一天，霍普正在哈佛大学教授艾肯(Howard H. Aiken)的指导下操作 Harvard Mark I 计算机。突然，一只倒霉的蛾子导致一个继电器失灵。霍普和其他程序员将这只死掉的蛾子粘到工作日志上，并添加了这样一行批注："First actual case of bug being found."(发现 bug 的第一个实际案例)。这个日志目前陈列于弗吉尼亚达尔格伦的中心博物馆。这是第一个被编入文档的计算机 bug。发生这个事件后，每当有人询问为什么还没有计算出结果，艾肯教授的人都会说正在为计算机 debug(除虫)。如果想进一步了解霍普和其他计算机历史人物，请阅读索尔特(Robert Salter)的 *Portraits in Silicon* 一书(MIT Press 1987 年出版)。今天，bug 被广泛地用于指代程序中的错误。本节将描述三种编程错误，并提供了对其进行纠正的一些建议。

各种程序错误

编译器能捕捉特定类型的错误，并在检测到错误后输出一条错误消息。它检测的是**语法错误**，表示程序违反了编程语言的语法规则，比如遗漏分号。

编译器发现程序含有语法错误，会指出错误位置及类型。如果编译器说有语法错误，你应该相信真的有语法错误。但是，它报告的错误位置和错误性质也许是不正确的。它能很好地将错误定位在一两行代码中，但不能很好地定位错误的来源。这是由于编译器会猜测你写的代码的意思，所以很容易猜错。毕竟，编译器无法读懂你的思想。在显示的一系列错误消息中，第一条之后的消息极有可能是不正确的(无论错误的位置还是性质)。同样地，这是由于编译器必须猜测你的意思。如果刚开始就猜错了，自然会影响到后面的分析。

如果程序包含的内容直接违反语言的语法规则，编译器会给出一条**错误消息**。但是，编译器有时只给出一条**警告消息**，表明代码从技术上说没有违反语法规则，但它出乎寻常，所以可能是一个错误。给出一条警告消息时，编译器相当于说："你真的是这个意思吗？"在程序开发的这一阶段，应该将每个警告都视为错误，除非教师允许忽略警告。

某些错误只有在程序运行时才会被计算机系统检测到，它们称为**运行时错误**。大多数计算机系统都能检测特定的运行时错误，并输出相应的错误消息。许多运行时错误与数值

---

① 参见 Denise W. Gurer 的"*Pioneering women in computer science*"一文，CACM 38(1):45-54 页，1995 年 1 月。

计算有关。例如，如果程序试图让一个数字除以 0，通常就会产生运行时错误。

即使编译器成功编译程序，而且程序运行一通后没有产生运行时错误，也不能保证程序就是正确的。记住，编译器只能告诉你是否写了一个语法正确的 C++程序，不能告诉你程序是否真的能做你希望它做的事情。基础算法的错误或者将算法翻译成 C++语言时的错误称为**逻辑错误**。例如在图 1.8 中，如果将乘号*错误地写成了加号+，就属于逻辑错误。程序虽然能编译和正常运行，但答案是错误的。如果编译器成功编译了程序，也没有产生运行时错误，但结果不正确，表明程序肯定存在逻辑错误。逻辑错误是最难诊断的一种错误，因为计算机不会提供任何帮助你定位这种错误的消息。这不是计算机的错，因为对计算机来说，它只是根据你写的代码来判断你的意图。

陷阱：错误地假定程序正确

为了测试一个新程序是否存在逻辑错误，应该使用几套有代表性的数据来运行程序，检查在各种输入下的表现。如果通过了测试，你就对它的正确性有了更大的信心，但这仍然不能保证程序是绝对正确的。用其他数据来运行时，它仍有可能表现异常。减少逻辑错误最好的办法就是防患于未然，编程时就应该非常仔细，这样能避免大多数错误。■

√ 自测题

22. 主要有哪三种程序错误？
23. 编译器能发现哪些类型的错误？
24. 如果在程序中遗漏了一个标点符号(比如一个分号)，就会产生一个错误。这是什么类型的错误？
25. 如果在程序中遗漏最后一个花括号}，就会产生一个错误。这是什么类型的错误？
26. 假定程序使编译器报告了一条警告消息。应该如何处理它？给出本书的标准答案；如果你的环境要求以不同方式处理，也给出相应的答案。
27. 假定要写一个程序计算银行账户的利息，假定账户利息每天都应该计算。但是，你的程序错误地每年计算利息。请问这是什么类型的程序错误？

小　结

- 计算机使用的各种程序称为这台计算机的**软件**。组装一台计算机所需的物理设备称为**硬件**。
- 计算机的 5 个主要部件是输入设备、输出设备、处理器(CPU)、主存储器和辅助存储器。
- 计算机有两种存储器：主存储器(内存)和辅助存储器(外存)。主存储器只在程序运行时使用。辅助存储器用于持久性地保存数据，程序运行之前和(或)之后，数据都会得以保留。
- 计算机的主存储器划分为一系列编号位置，这些位置称为**字节**。与每个字节对应的编号就是那个字节的地址。通常，几个字节组合成一个较大的内存位置。在这种情况下，第一个字节的地址就成为这个较大的内存位置的地址。
- **字节**由 8 个二进制位构成，每个二进制位要么为 0，要么为 1。只能为 0 或 1 的数位称为一个**比特**或者直接称为**位**。
- **编译器**是一种特殊的程序，它将用高级语言(比如 C++)写成的程序翻译成机器语言程序，后者可由计算机直接理解和执行。
- 作为解决方案的一系列准确的指令称为一个**算法**。算法可以用自然语言来写，或者用 C++这样的编程语言来写。但是，“算法”一词通常表示用中文(或其他自然语言，比如西班牙语或者英语等)写成的一系列指令。
- 写 C++程序之前，要先设计好程序使用的算法(解决方案)。
- 有三类程序错误：语法错误、运行时错误和逻辑错误。计算机通常能指出前两类错误。但逻辑错误必须由你自己去发现。
- C++程序中的单独一个指令称为**语句**。
- C++程序中的变量可用于命名一个数字(变量的详情在第 2 章解释)。
- C++程序中，以 `cout <<`开头的语句是输出语句，它告诉计算机将<<之后的内容输出到屏幕。
- C++程序中，以 `cin >>`开头的语句是输入语句。

自测题答案

1. 计算机的 5 个主要部件是输入设备、输出设备、处理器(CPU)、主存储器和辅助存储器。
2. 要相加的 2 个数。
3. 每次考试和每次作业时的每个学生的成绩。
4. 机器语言程序采用计算机能直接执行的形式写成。高级语言程序采用人们容易阅读的形式写成。高级语言程序必须在转换成机器语言程序之后，才能由计算机执行。
5. 编译器将高级语言程序转换成机器语言程序。
6. 输入编译器的高级语言程序称为源程序。由编译器输出的、转换好的机器语言程序称为目标程序。
7. 操作系统是一个程序，或者是相互协作的多个程序，但你最好把它想象成自己的管家。
8. 操作系统的用途是为计算机必须完成的任务分配计算机的资源。
9. 可能的答案是 Mac OS、Windows 2000、Windows XP、VMS、Solaris、SunOS、UNIX(或者类 UNIX 操作系统，如 Linux)等。
10. C++程序的目标码必须与程序所用例程(比如输入和输出例程)的目标码合并起来。这个合并目标码的过程称为链接。对于简单程序，这个链接过程是自动完成的。
11. 取决于你的编译器，答案也会不同。大多数 UNIX 和类 UNIX 风格的编译器都是自动链接的，Windows 和 Macintosh 操作系统的大多数集成开发环境(IDE)所提供的编译器也是这样的。
12. 以下指令过于含糊，不能用在算法中：
    - 根据口味添加香草精
    - 搅拌均匀
    - 倒入一个漂亮的玻璃杯
    - 撒上肉豆蔻

    说明：“根据口味”、“均匀”和“好看”等说法都是不精确的。“撒”的指示也过于含糊，因为它没有指定具体要撒多少肉豆蔻。其他指令都是合理的，可在一个算法中使用。
13. 创建一个程序时，第一步是确定程序要完成的任务已经得到完整和准确的定义。
14. 问题求解阶段和实现阶段。
15. 经验表明，分两阶段进行的程序设计过程能更快地生成一个能正确工作的程序。
16. `C++ is easy to understand.`
17. 符号\n 告诉计算机在输出中开始一个新行，使以后的输出从下一行开始。
18. 这个语句告诉计算机读取键盘输入的下一个数字，并将那个数字发送给名为 peasPerPod 的变量。
19. 这个语句表示要让变量 numberOfPods 和 peasPerPod 中的数字相乘，将乘积放到名为 totalPeas 的变量中。
20. #include <iostream>预编译指令告诉编译器获取文件 iostream。该文件包含了提供了 I/O 功能的 cin、cout、插入(<<)和提取(>>)操作符的声明。这样就能将来自 iostream 库的目标码与程序中的 I/O 语句正确链接。
21. a. iostream 文件名之后多余的空格导致“文件未找到”错误消息。
    b. iostream 文件名之前多余的空格导致“文件未找到”错误消息。
    c. 正确。
22. 三种主要的程序错误是语法错误、运行时错误和逻辑错误。

23. 编译器检测语法错误。一些错误从技术上说不是语法错误，但暂时把它们归为语法错误。你以后会更详细地了解它们。
24. 语法错误。
25. 语法错误。
26. 本书要求将警告视为错误。但在你的学习环境中，应该询问教师，了解如何处理警告消息。
27. 逻辑错误。

编 程 练 习

编程练习一般只需写很短小的程序，运用本章提到的编程概念。

1. 用文本编辑器录入图 1.8 的 C++程序。一定要完全按照图 1.8 的样子录入第一行。尤其是第一行必须从最左侧开始，而且符号#前后没有多余空格。编译并运行程序。如果编译器给出错误消息，请改正程序并重新编译。一直进行这个操作，直到编译器不再报错。最后运行程序。
2. 修改编程项目 1 录入的 C++程序，使它首先将单词“Hello”写到屏幕上。程序其余的操作不做改动，完全和图 1.8 保持一致。在程序中添加一行就可达到目的。重新编译程序，运行修改过的程序。然后，进一步修改程序。再添加一行代码，使程序在结束之前将单词“Good-bye”输出到屏幕上。一定要为最后一个输出语句添加符号\n：

   ```
   cout << "Good-bye\n";
   ```

   (有的系统要求必须最后添加一个\n，你的系统可能正好有此要求)。重新编译并运行更改过的程序。
3. 继续修改上一题的 C++程序。将乘号(*)变成除号(/)。重新编译修改好的程序。运行程序。在提示“number of peas in a pod”(一个豆荚中的豆子数)时输入 0。注意除以 0 所引起的运行时错误消息。
4. 修改编程项目 1 录入的 C++程序。将程序中的乘号*改成加号+。重新编译并运行修改过的程序。注意程序会正常编译和运行，只是输出不正确。这是因为这一处修改造成了逻辑错误。
5. 修改编程项目 1 录入的 C++程序，计算用于封闭宽度为 width、长度为 height 的一个长方形区域所需的篱笆总长度。程序要求用户输入 width 和 height 的值。再创建第三个变量 totalLength 来存储计算得到的篱笆总长度。在一条恰当的消息中输出该值。
6. 视频讲解：*Solution to Practice Program 1.6*

   这个练习的目的是生成初学者可能遇到的典型语法错误和错误消息的一个目录，并促使学生熟悉自己的编程环境。通过本练习，学生可在遇到常见的错误消息时，在自己的程序中准确找出相应的错误。你的老师可能已为本练习准备了一个实验程序。如果没有，请从前面的编程项目中选用一个程序。你需要在程序中故意犯一个错误，编译程序，记录错误和错误消息，改正错误，并重新编译(确定程序得到改正)，再试验下一个错误。请制作一个错误目录，并在以后遇到新的程序错误和消息时，对目录进行增补。

   建议试验以下错误。
   a. 在<和 iostream 文件名之间插入一个多余的空格。
   b. 在 include 预编译指令中，省略一个<或>符号。
   c. 在 int main()中略去 int。
   d. 略去或误拼单词 main。
   e. 略去 main 之后的圆括号之一，也就是 (或)。然后，将两个圆括号都略去。
   f. 继续试验其他错误，故意拼错标识符(cout、cin 等)。在 cout 语句中略去<<，略去结束花括号}等。

编 程 项 目

编程项目要求综合运用多方面的知识来解决问题，程序一般比编程练习大，解题方式多样化。

1. 写 C++程序读入两个整数，输出两个数的和与积。一个办法是以图 1.8 的程序为基础，修改它，把它变成适于本项目的程序。第一行必须和图 1.8 一样。尤其是第一行必须从最左侧开始，而且#符号前后没有多余空格。另外，为程序最后一个输出语句添加符号\n。例如，最后一个输出语句可能如下：

   ```
   cout << "This is the end of the program.\n";
   ```

   (有的系统要求必须在最后添加\n，你的系统可能正好有此要求)。

2. 写程序打印 CS!图案。图案包含在由*构成的两条线之间，后跟两个空行，然后打印消息“Computer Science is Cool Stuff!!!”。下面是最终的输出结果：

   ```
   *****************************************************

             C C C             S S S S        !!
           C       C         S         S      !!
          C                 S                 !!
         C                   S                !!
         C                    S S S S         !!
         C                             S      !!
          C                             S     !!
           C       C         S         S
             C C C             S S S S        00

   *****************************************************
   Computer Science is Cool Stuff!!!
   ```

3. 视频讲解：*Solution to Programming Project 1.3*

   写程序要求用户输入 25 美分、10 美分和 5 美分硬币枚数，以美分为单位输出所有硬币的币值。例如，如果输入 2 表示 25 美分硬币的枚数，3 表示 10 美分硬币的枚数，1 表示 5 美分硬币的枚数，程序应输出币值为 85 美分。

4. 写程序要求用户输入一个秒数。程序输出物体自由落体那个时间的距离。假定物体最初静止，忽略空气摩擦力或阻力，而且重力加速度固定为每平方秒 32 英尺(9.8 米)。公式如下：

   $$距离=\frac{加速度\times时间^2}{2}$$

   应该先计算乘积，再将结果除以 2(本书以后会讨论为什么要这样做)。

5. 写程序从键盘输入一个字符，输出由该字符构成的大型字母 C。例如，输入“X”将产生以下输出：

   ```
     X X X
    X     X
   X
   X
   X
   X
   X
    X     X
     X X X
   ```

第 2 章　C++基础知识

别以为你知道计算机终端是个什么东西。计算机终端可不是什么乏味的旧电视，前头再摆个打字机。它是一种接口，使身体和心灵可以和宇宙相连接，并且把其中的一些东西移来移去。

——道格拉斯·亚当斯，《银河系漫游指南》第五卷·基本无害

概述

本章将解释更多的 C++示范程序，展示 C++语言足够多的细节，便于你写出简单的 C++程序。

预备知识

第 1 章简单介绍了一个 C++示范程序，本章将使用那个程序(如果还没有阅读对那个程序的描述，请在继续后面的学习之前阅读它，这对你很有帮助)。

2.1 变量和赋值

一旦理解变量在编程中的用法，就可以说理解了编程的精髓。

——艾兹格·戴克斯特拉，“结构化编程”课堂笔记

程序要处理数字和字母之类的数据。C++和其他常用编程语言一样，使用名为**变量**的编程构造来命名和存储数据。变量是编程语言(如 C++)的核心，所以要从变量开始介绍 C++。下面将围绕图 2.1 的程序展开讨论，并解释该程序中的所有元素。虽然此程序的常规思路应该是很清楚的，但某些细节是新的，需要进行一些解释。

变量

C++变量可容纳一个数字或其他类型的数据。目前只关心用于存储数字的变量。这些变量类似于可在上面写数字的小黑板。黑板上写的数字可以更改，C++变量容纳的数字也可以。不过，黑板可能不包含任何数字，但变量肯定包含了某个值——可能是以前运行过的程序在内存中留下的垃圾数字。变量容纳的数字或其他类型的数据称为这个变量的**值**。也就是说，变量值是在一个假想的黑板上写下的内容。图 2.1 中，numberOfBars，oneWeight 和 totalWeight 是变量。例如，如果运行程序并提供示范对话中的输入，numberOfBars 的值会被设为 11，这是通过以下语句来实现的：

```
cin >> numberOfBars;
```

之后，当程序执行以上语句的另一个拷贝时，变量 numberOfBars 的值会被更改为 12。稍后会详细解释这是怎样发生的。

图 2.1　一个 C++程序

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5      int numberOfBars;     // 糖的数量
6      double oneWeight, totalWeight; // 每块糖的重量和总的重量
7
8      cout << "Enter the number of candy bars in a package\n"; // 输入一包糖中有多少块糖
9      cout << "and the weight in ounces of one candy bar.\n"; // 每块糖的重量(盎司)
10     cout << "Then press return.\n"; // 输入后按 Enter 键
11     cin >> numberOfBars;
12     cin >> oneWeight;
13
14     totalWeight = oneWeight * numberOfBars;
15
16     cout << numberOfBars << " candy bars\n";
17     cout << oneWeight << " ounces each\n";
18     cout << "Total weight is " << totalWeight << " ounces.\n";
19
20     cout << "Try another brand.\n";
21     cout << "Enter the number of candy bars in a package\n";
22     cout << "and the weight in ounces of one candy bar.\n";
23     cout << "Then press return.\n";
24     cin >> numberOfBars;
25     cin >> oneWeight;
26
27     totalWeight = oneWeight * numberOfBars;
28
29     cout << numberOfBars << " candy bars\n";
30     cout << oneWeight << " ounces each\n";
31     cout << "Total weight is " << totalWeight << " ounces.\n";
32
33     cout << "Perhaps an apple would be healthier.\n";
34
35     return 0;
36 }
```

示范对话

```
Enter the number of candy bars in a package and the weight in
ounces of one candy bar.
Then press return.
11 2.1
11 candy bars
2.1 ounces each
Total weight is 23.1 ounces.
Try another brand.
Enter the number of candy bars in a package and the weight in
ounces of one candy bar.
Then press return.
12 1.8
12 candy bars
1.8 ounces each
Total weight is 21.6 ounces.
Perhaps an apple would be healthier.
```

当然，变量不是黑板。在编程语言中，变量作为内存位置来实现。编译器将内存位置(参见第 1 章的讨论)分配给程序中的每个变量名。0 和 1 形式的变量值存储在为变量分配的内存位置。例如，在图 2.1 的程序中，为这三个变量分配的内存位置可能分别是 1001，1003 和 1007。具体编号取决于计算机、编译器和其他很多因素。我们不知道、也不关心编译器为变量选择什么内存地址。可以简单地认为内存位置就是用变量名作为标签。

> **程序无法运行**
>
> 如果无法编译和运行 C++程序，请阅读 1.3.8 节了解如何应对不同的 C++编译器和 C++环境。

名称：标识符

在示范程序中，首先注意到的可能是变量名比平时在数学课上使用的变量名长。为了使程序容易理解，务必为变量使用有意义的名称。变量(或者在程序中定义的其他项目)的名称叫**标识符**。标识符以字母或下划线开头，其余字符必须是字母、数字或下划线。例如，下面的标识符是有效的：

```
x x1 x_1 _abc ABC123z7 sum RATE count data2 Big_Bonus
```

编译器会接受这些标识符，但前 5 个不太理想，它们没有表达出用途。以下标识符则是无效的，编译器拒绝接受：

```
12 3x %change data-1 myfirst.c PROG.CPP
```

前 3 个之所以无效，是因为不是以字母或下划线开头。其余 3 个之所以无效，是因为包含了除字母、数字和下划线之外的其他符号。

C++是**对大小写敏感**的语言。也就是说，它会区别对待标识符中的大写和小写字母。因此，以下 3 个标识符是不同的标识符，可命名 3 个不同的变量：

```
rate RATE Rate
```

但最好不要在同一个程序中使用这样的变体，因为它们太容易混淆。虽然 C++没有专门要求，但变量名最好全部小写。预定义标识符(比如 main，cin 和 cout 等)则必须全部小写。本章后面会讲到一些采用大写字母的标识符。

C++标识符长度没有限制，但有的编译器设置了最大允许长度，超出的会被忽略。

> **标　识　符**
>
> **标识符**用于命名 C++程序中的变量和其他元素。标识符必须以字母或下划线开头，后续每个字符只能是字母、数字或下划线。

还有一类特殊标识符称为**关键字**或**保留字**，它们在 C++中有预定义的含义，不能用作变量或其他元素的名称。附录 1 列出了所有 C++关键字。

你可能会问，为什么定义成 C++语言一部分的其他单词未被归为关键字？cin 和 cout 等单词为什么不是关键字？原来，程序员可以重新定义这些单词(虽然这样容易使人混淆)。但这些预定义的单词确实不是关键字，它们是在 C++语言标准要求的库中定义的。本书后面会讨论库，目前暂时不必关心库的问题。毫无疑问，为预定义标识符赋予非标准的含义，肯定会产生误导，而且很危险，所以应尽量避免。最安全、最简单的做法是将所有预定义标识符也视为关键字。

变量声明

C++程序中的每个变量都必须声明。**声明**变量实际是告诉编译器(最终是告诉计算机)：准备在该变量中存储什么类型的数据。例如，图 2.1 用两个语句声明了三个变量：

```
int numberOfBars;
double oneWeight, totalWeight;
```

在一个语句中声明多个变量要用逗号分隔不同的变量。另外，每个语句以分号结尾。

上面两个声明语句中，第一个语句中的 int 是 integer(整数)一词的缩写(但在 C++程序里，必须使用缩写形式 int，千万不能写全称)。这行代码将标识符 numberOfBars 声明为 int 类型的变量。这表示 numberOfBars 的值必须是整数，比如 1，2，-1，0，37 或-288。

第二个语句中的 double 将两个标识符 oneWeight 和 totalWeight 声明为 double 类型的变量。double 类型的变量可存储带小数部分的值，如 1.75 或-0.55。变量能容纳的数据的种类称为这个变量的**类型**，类型的名称(如 int 或 double)称为**类型名称**。

变 量 声 明

所有变量必须在使用之前声明。变量声明语法如下：

语法

Type Name VariableName1, *VariableName2*, ...;

示例

```
int count, numberOfDragons, numberOfTrolls;
double distance;
```

命 名 规 范

命名规范是指标识符(比如变量名)的一套命名规则。本书变量名以小写字母开头。如变量名由多个单词复合，之后各单词以大写字母开头。这个规范称为 camelCase。C 风格的规范要求各单词以下划线连接。还有的规范要求在变量名中指定变量类型。

C++程序的每个变量都必须在使用前声明。有两个非常自然的位置可供声明变量：刚好在使用变量之前，或者在程序 main 部分的起始处，也就是在以下两行代码之后：

```
int main()
{
```

总之，选择能使程序变得更清晰的位置。

变量声明提供了编译器实现变量所需的信息。前面说过，编译器将变量实现为内存位置，而变量的值保存在为这个变量分配的内存位置中。变量的值被编码为一连串 0 和 1。不同类型的变量需要不同大小的内存位置，还需要用不同的编码方式把它们的值编码为一连串 0 和 1。计算机用一种方式将整数编码为一连串 0 和 1，用另一种方式编码带有小数部分的数字，再用另一种方式将字母编码为一连串 0 和 1。变量声明实际是告诉编译器(最

终是告诉计算机)两点：第一，应该为一个变量分配多大的内存位置；第二，用哪种编码方式将变量的值表示为一连串 0 和 1。

> **语法和语义**
>
> 编程语言(或其他任何语言)的**语法**(syntax)指该语言的一套语法规则。例如，谈论变量声明的语法时(参见上一个小结框"变量声明")，实际是说为了写一个具有良好格式的变量声明，需要遵循哪些规则。遵循了 C++的所有语法规则，编译器就肯定接受你的程序。当然，这只能保证程序有效。虽然能保证程序能做某事，但不能保证它做的是你真正想做的事情。
>
> **语义**(semantics)是指程序运行时所做事情的含义。程序的语法和语义都应正确。

赋值语句

更改变量值最直接的方式就是使用赋值语句。**赋值语句**是表示"将变量设为指定值"的计算机指令。以下赋值语句摘自图 2.1 的程序：

```
totalWeight = oneWeight * numberOfBars;
```

它要求计算机将 totalWeight 的值设为两个变量(oneWeight 和 numberOfBars)中的数字的乘积(第 1 章讲过，*是 C++中的乘号)。

赋值语句总是由等号左侧的变量和等号右侧的表达式组成。赋值语句以分号结尾。等号右侧的表达式可以是变量、数字或者由变量、数字和算术操作符(如*和+)构成的较复杂的表达式。赋值语句指示计算机对等号右侧的表达式进行求值(计算这个表达式的值)，并把等号左侧的变量的值设为求值的结果。多研究一些例子有助于掌握赋值语句的工作原理。

可用任何算术操作符代替乘号(*)。例如，以下语句也是有效的赋值语句：

```
totalWeight = oneWeight + numberOfBars;
```

该赋值语句与示范程序的赋值语句一样，区别在于执行加法而非乘法。它将 totalWeight 的值设为两个变量值(oneWeight 和 numberOfBars)之和。当然，如果在图 2.1 中进行这样的改动，程序虽会运行，但会给出不正确的输出。

赋值语句的等号右侧的表达式可以是另一个变量。以下语句使变量 totalWeight 的值变得和变量 oneWeight 一样：

```
totalWeight = oneWeight;
```

在图 2.1 的程序中进行上述修改，程序给出的一包糖的重量将错误地小于实际值(假定一包糖内不止一块糖)。但在其他程序中，这样赋值可能是有意义的。

作为另一个例子，以下赋值语句将 numberOfBars 的值变成 37：

```
numberOfBars = 37;
```

代码中的实际数字(如本例的 37)称为**常量**。和变量不同，常量的值不能改变。

由于变量值可以随时间更改，同时赋值操作符是更改其值的手段，所以赋值语句的含义必须考虑到时间因素。在时间上，首先求值的是等号右侧的表达式。然后，等号左侧的

变量的值被更改为那个表达式的求值结果。这就意味着一个变量可以同时出现在赋值操作符的两侧。例如以下赋值语句：

```
numberOfBars = numberOfBars + 3;
```

初学者可能不解其意。如果像普通的句子那样读，就是“numberOfBars 等于 numberOfBars 加 3。”但实际含义是“让 numberOfBars 的新值等于 numberOfBars 的旧值加 3。”在 C++中，等号的用法与日常语言或数学中的等号不同。

赋 值 语 句

赋值语句中，等号右侧的表达式先求值，该值赋给等号左侧的变量。

语法

```
Variable = Expression;
```

示例

```
distance = rate * time;
count = count + 2;
```

陷阱：未初始化的变量

除非程序为变量赋值，否则变量不包含有意义的值。例如，如果变量 minimumNumber 既没有放在一个赋值语句的左侧而被赋值，也没有通过其他手段来赋值(比如用 cin 语句赋一个由用户输入的值)，以下语句就是错误的：

```
desiredNumber = minimumNumber + 10;
```

这是由于 minimumNumber 不包含有意义的值，所以等号右侧的整个表达式也不会产生有意义的值。如变量(比如 minimumNumber)未被赋值，就说它**未初始化**。事实上，这比 minimumNumber 不包含任何值还要糟。未初始化的变量(比如 minimumNumber)将包含“垃圾值”。未初始化的变量的值由留在其内存位置中的 0，1 序列来决定(可能是由用过该内存位置的上一个程序留下的)。因此，如果程序运行两次，未初始化的变量每次都可能获得不同的值。只要程序为完全相同的输入数据产生了两个不同的输出，而且程序本身未进行任何修改，就应该怀疑其中含有未初始化的变量。

为了避免出现未初始化的变量，一个办法是在声明变量的同时初始化。这可通过添加一个等号和一个值来完成，如下所示：

```
int minimumNumber = 3;
```

它除了将 minimumNumber 声明为 int 类型的变量，还将 minimumNumber 的值设为 3。像这样在同一行声明并初始化时，可以使用涉及运算(比如加法或乘法运算)的一个更复杂的表达式。然而，最常见的还是使用简单的常量来初始化。可在一个声明内列出多个变量，同时初始化其中的部分或全部变量，或者一个都不初始化。例如，以下语句声明了三个变量，但只初始化其中两个：

```
double rate = 0.07, time, balance = 0.0;
```

C++允许采用另一种方式在声明变量的同时初始化它。以下语句等价于刚才的声明：

```
double rate(0.07), time, balance(0.0);
```

具体是在声明的同时初始化，还是以后再初始化，要取决于实际情况。你选择的方式应该使程序更容易阅读。■

声明时初始化变量

可在声明变量的同时初始化它(向它赋值)。

语法

Type_Name　variableName1 = Expression_for_Value_1,
variableName2 = Expression_for_Value_2, . . .;

示例

```
int count = 0, limit = 10, fudgeFactor = 2;
double distance = 999.99;
```

声明变量同时初始化的另一种方式：

Type_Name variableName1(Expression_for_Value_1),
variableName2(Expression_for_Value_2), . . .;

示例

```
int count(0), limit(10), fudgeFactor(2);
double distance(999.99);
```

编程提示：使用有意义的名称

变量名和其他名称至少应该让人联想到它们所命名的事物的含义或用途。有意义的变量名可以使程序更容易理解。所以，以下语句：

```
x = y * z;
```

最好更改为：

```
distance = speed * time;
```

两个语句作用一样，但第二个更容易理解。

√ 自测题

1. 给出变量 feet 和变量 inches 的声明。两个变量都是 int 类型，都在声明中初始化为 0。请同时使用前面介绍的两种初始化方式。
2. 给出变量 count 和变量 distance 的声明。count 是 int 类型，初始化为 0；distance 是 double 类型，初始化为 1.5。
3. 给出 C++语句，将变量 sum 的值更改为变量 n1 和 n2 的值之和。所有变量都是 int 类型。
4. 给出 C++语句，使变量 length 的值增大 8.3。变量 length 已经声明为 double 类型。
5. 给出 C++语句，将变量 product 的值更改为它的旧值乘以变量 n 的值。这些变量都是 int 类型。
6. 写程序来输出虽已声明但尚未初始化的五六个变量的值。编译并运行。会输出什么？请说明原因。

7. 为以下每个变量提供有意义的名称：
   a. 一个用于保存车速的变量；
   b. 一个用于保存小时工每小时薪酬的变量；
   c. 一个用于保存一次考试的最高分的变量。

2.2 输入和输出

垃圾入，垃圾出。

——*程序员的口头禅*

C++程序可采取几种不同的方式执行输入和输出。我们描述其中的一种，即“流”。**输入流**是提供给计算机并由程序使用的一系列输入。“流”这个词表明程序将以相同的方式处理所有输入，无论这些输入是从什么地方来的。换言之，程序看到的只是输入流本身，看不到流的来源。好比山间一条小溪，溪水在你面前流过，而你不知道它来自哪里。本节假设输入来自键盘。第 6 章将讨论程序如何从文件读取输入。到时就会知道，从键盘读取输入的语句也可用于从文件读取。类似地，**输出流**是程序生成的一系列输出。本节假设输出到终端屏幕(第 6 章将讨论如何输出到文件)。

使用 cout 进行输出

可以使用 cout 将变量值和文本字符串输出到屏幕。变量和字符串可组合输出。以图 2.1 的下面这行代码为例：

```
cout << numberOfBars << " candy bars\n";
```

它要求计算机输出两项内容：变量 numberOfBars 的值和用引号封闭的字符串" candy bars\n"。注意，不需要为每个输出项都单独使用单词 cout。可以列出所有输出项，在每一项之前附加箭头符号<<。上述 cout 语句等价于以下两个 cout 语句：

```
cout << numberOfBars;
cout << " candy bars\n";
```

cout 语句可以包括算术表达式，如下例所示，其中的 price 和 tax 是变量：

```
cout << "The total cost is $" << (price + tax);
```

用于封闭算术表达式(如 price + tax)的圆括号是一些编译器要求的，最好不要遗漏。

两个<(小于)符号应连续输入，中间不要有空格。箭头符号<<通常称为**插入操作符**。整个 cout 语句以分号结尾。

只要连续出现了两个 cout 语句，就可以将其合并成一个较长的 cout 语句。以图 2.1 的以下两行代码为例：

```
cout << numberOfBars << " candy bars\n";
cout << oneWeight << " ounces each\n";
```

这两个语句可改写为如下所示的一个语句，程序执行结果一样：

```
cout << numberOfBars << " candy bars\n" << oneWeight << " ounces each\n";
```

为了防止代码行超出屏幕边界，可将较长的 cout 语句分解为两行或更多的行。一种较好的做法是将上述较长的 cout 语句改写成如下形式：

```
cout << numberOfBars << " candy bars\n"
     << oneWeight << " ounces each\n";
```

不要在引号字符串中断行。但是，凡是能插入空格的地方都能另起一行。计算机接受任何合理的间隔与换行风格，但上例和本书的其他示范程序是你应该学习的“好榜样”。一个较好的策略是，针对直观上可视为一个整体的每组输出都使用一个 cout。注意，每个 cout 都只对应一个分号，即使一个 cout 语句被拆分成若干行。

在图 2.1 的程序中，特别注意要输出的引号字符串。注意，要输出的字符串必须包含在一对双引号内。每个双引号都是一个单独的字符(按一次键即可输入)，不要连续键入两个单引号来取代它。还要注意，字符串两端使用的是同一个双引号，没有独立的左引号和右引号。

还要注意引号内的空格。计算机不会在 cout 输出的内容前后自动添加空格。因此，示例中的引号字符串通常都会以一个空格开始和/或结束。空格防止不同的字符串和数字紧挨在一起。如果只想输出空格，就使用只包含空格的字符串。如下所示：

```
cout << firstNumber << " " << secondNumber;
```

如第 1 章所述，\n 告诉计算机从一个新行输出。除非告诉计算机换行，否则它会将所有输出放到同一行。取决于计算机屏幕设置，这可能导致输出时任意换行[①]，或者一行的剩余内容跑到屏幕外面。注意\n 是在引号内部使用的。在 C++中，“换行”被视为一个特殊字符(特殊符号)。在引号字符串内，这个特殊字符要拼写成\n。注意，符号\和 n 之间没有空格。虽然这个特殊字符在输入时要使用两个符号，但 C++将\n 视为一个字符，并称之为**换行符**。

include 预编译指令和命名空间

我们所有程序都以下面这两行代码开始：

```
#include <iostream>
using namespace std;
```

这两行使 iostream 库进入可用状态。cin 和 cout 的定义就包含在这个库中。所以，假如程序使用了 cin 或 cout，就应在程序文件起始位置包含上述两行代码。

下面这一行代码称为 include **预编译指令**。它将 iostream 库“包含”到程序中，使程序能使用 cin 和 cout：

```
#include <iostream>
```

cin 和 cout 在一个名为 iostream 的文件中定义，以上 include 指令相当于将那个文件复制到程序中。第二行代码比较复杂，三言两语很难说清。

C++使用**命名空间**组织名称。命名空间是很多名称(比如 cin 和 cout)的集合。通过以下方式指定命名空间的语句称为 using **预编译指令**：

① 例如，单词“word”的“ord”部分可能跑到下一行。——译注

```
using namespace std;
```

这个特定的 using 指令表明程序准备使用 std(指 standard)命名空间。这意味着你使用的名称具有 std 命名空间为其定义的含义。这儿的重点在于，cin 和 cout 等名称在 iostream 中定义时，它们的定义指出它们在 std 命名空间中。所以，要使用 cin 和 cout 等名称，就要告诉编译器你准备“using namespace std;”。

对于命名空间并不需要了解太多(就目前而言)，但简单澄清一下，有助于解除你在使用命名空间时可能产生的一些困惑。C++之所以有命名空间，是因为有太多的东西需要命名。结果就是可能有两个或更多的项同名。换句话说，一个名称可能具有两个不同的定义。为消除歧义，C++将不同的项划分到不同的集合中，确保同一个集合(即同一个命名空间)中没有任何两个项同名。

注意，命名空间并不只是一个名称集合。它代表了一个 C++代码主体，其中指定了某些名称的含义(比如一些定义和/或声明)。命名空间的作用是将所有 C++名称规范划分成不同的集合(称为命名空间)，使命名空间内的每个名称在那个命名空间中都只有一个“规范”(一个定义)。命名空间对名称进行划分，但和那些名称配合的还有大量 C++代码。

可以使用两个命名空间中的两个同名元素吗？答案是肯定的，而且并不复杂，但那是本书后面要讲的一个主题。目前不需要这样做。

有的C++版本使用 include 预编译指令的一种古老形式(没有任何 using namespace)：

```
#include <iostream.h>
```

使用以下语句时，如果程序无法编译或者不能运行：

```
#include <iostream>
using namespace std;
```

请尝试将上述两行代码统一更换为如下形式：

```
#include <iostream.h>
```

但如果程序要求使用 iostream.h(而不是 iostream)，则表明你使用的是老版本的 C++编译器。升级一下吧。

转义序列

字符前的符号\告诉编译器：\之后的字符具有特殊含义，不能沿用其字面含义。这样的一个字符序列称为**转义序列**。转义序列肯定由两个字符构成，而且两个字符之间没有空格。C++定义了几个转义序列。

如果希望在一个字符串常量中插入反斜杠\或插入双引号"，则必须使用\"来转变"的原有功能(结束一个字符串常量)，或者使用\\来转变\的原有功能(转义)。\\向编译器表明你需要一个真正的反斜杠\，而不是一个转义序列。\"表明需要一个真正的双引号，而不是结束一个字符串常量。

如果在字符串常量中出现未定义的转义序列(比如\z)，有的编译器会返回一个 z，有的则会报错。ANSI 标准规定，对于未定义的转义序列，其行为是“未定义”的。所以，编译器的设计者可采用自己觉得方便的任何方式处理它。其后果是，如果在代码中使用了未定义的转义序列，就失去了“可移植性”。因此，不要使用任何未定义的转义序列。下面列

出了 C++定义的部分转义序列：

- 换行符　　\n
- 水平制表符　\t
- 响铃符　　\a
- 反斜杠　　\\
- 双引号　　\"

此外，C++11 支持所谓的**原始字符串字面值**(raw string literals)，它适合有太多字符需要转义的情况。该格式要求字符串以 R 开头，而且字符串内容要放到一对圆括号中。例如，以下代码输出字符串字面值"c:\files\"。

```
cout << R"(c:\files\)";
```

要在输出中插入空行，可单独输出一个\n 换行符，如下所示：

```
cout << "\n";
```

输出空行的另一种方式是使用 endl，含义和"\n"相同，所以还能像下面这样输出空行：

```
cout << endl;
```

虽然"\n"和 endl 含义相同，但用法稍有区别：\n 必须放到双引号内，endl 则不能。

\n 和 endl 应该如何选择呢？一个较好的依据是：如果\n 可以放到一个较长的字符串的末尾，就像下面这样使用\n：

```
cout << "Fuel efficiency is "
     << mpg << " miles per gallon\n";
```

而如果需要单独使用一个"\n"，就改为使用 endl，如下所示：

```
cout << "You entered " << number << endl;
```

在输出中开始新行

要在输出中换行，可将\n 包含到引号字符串内，如下所示：

```
cout << "You have definitely won\n"
<< "one of the following prizes:\n";
```

记住，\n 要作为两个字符来输入，两个字符之间不能有空格。

另一种方法是通过 endl 来换行，如下所示：

```
cout << "You have definitely won" << endl
<< "one of the following prizes:" << endl;
```

编程提示：用\n 或 endl 终止每一个程序

最好在每个程序末尾都输出一个换行指令。如果程序输出的最后一项是字符串，就在该字符串的末尾添加一个\n；如果不是，就将输出 endl 作为程序的最后一个动作。这样做有两个目的：有的编译器不输出程序中的最后一行，除非在末尾包含一个换行指令。在另一些系统上，即使没有这个最后的换行指令，程序也能正常运行，但运行下一个程序时，它会将自己的第一行输出与上一个程序的最后一行输出混在一起。即使这两个问题在你的

系统上都不存在，在程序末尾添加一个换行指令，起码也能使程序具有更强的移植性。

格式化带小数点的数字

计算机输出 double 类型的值时，格式可能与你预期的不同。例如，下面这个简单的 cout 语句就会产生很多种不同的输出：

```
cout << "The price is $" << price << endl;
```

如果 price 的值是 78.5，输出结果可能如下：

```
The price is $78.500000
```

或者如下：

```
The price is $78.5
```

甚至可能是以下形式(详情参见 2.3 节)：

```
The price is $7.850000e01
```

以下输出恐怕是最不可能出现的，但这种格式才是最有意义的：

```
The price is $78.50
```

为保证得到预期输出，要在程序中包含一些指令告诉计算机如何输出数字。

可以在程序中插入一个“魔法配方”，指定含有小数点的数字(比如 double 类型的数字)以日常生活中习惯的方式来输出。换言之，可以确切地指定小数位数。例如，为了显示两位小数，需要以下“魔法配方”：

```
cout.setf(ios::fixed);
cout.setf(ios::showpoint);
cout.precision(2);
```

在程序中插入上述 3 个语句，后续的任何 cout 语句都会按你指定的格式输出 double 类型的值。换言之，小数点之后刚好 2 位。例如，假设以下 cout 语句出现在上述“魔法配方”之后的某个地方，并假设 price 的值为 78.5：

```
cout << "The price is $" << price << endl;
```

则输出结果肯定如下：

```
The price is $78.50
```

可用任何非负的整数代替“魔法配方”中的 2，从而指定不同的小数位数。甚至能用 int 类型的变量代替 2。

这个“魔法配方”将在第 6 章进一步解释。现在只需将其视为一长串指令，目的是告诉计算机你想以什么格式输出含小数点的数字。

要更改小数位数，使程序中不同的值输出不同的小数位数，可以重复使用这个“魔法配方”，用其他数字代替 2。但是，重复这些“魔法配方”时，只需重复其中的最后一行。假如该“魔法配方”已在程序中出现过一次，那么只需添加下面这一行，即可让后续的所有 double 类型的变量输出 5 位小数：

```
cout.precision(5);
```

输出 double 类型的值

在程序中插入以下“魔法配方”，后续所有 double 类型(或允许保留小数的其他类型)的数字将以日常生活中最习惯的方式来输出，即保留两位小数：

```
cout.setf(ios::fixed);
cout.setf(ios::showpoint);
cout.precision(2);
```

可用其他任何非负的整数代替其中的 2，从而指定不同的小数位数。甚至能用 int 类型的变量代替 2。

用 cin 进行输入

使用 cin 输入时，方式与使用 cout 输出差不多。两者语法相似，区别在于 cin 代替了 cout，而且箭头方向相反。例如，在图 2.1 的程序中，numberOfBars 和 oneWeight 变量由以下 cin 语句来填充(同时给出了 cout 语句，告诉用户应该如何操作)：

```
cout << "Enter the number of candy bars in a package\n";
cout << "and the weight in ounces of one candy bar.\n";
cout << "Then press return.\n";
cin >> numberOfBars;
cin >> oneWeight;
```

可以在一个 cin 语句中列出多个变量。所以，以上代码可以重写为以下形式：

```
cout << "Enter the number of candy bars in a package\n";
cout << "and the weight in ounces of one candy bar.\n";
cout << "Then press return.\n";
cin >> numberOfBars >> oneWeight;
```

如果愿意，还可将上述 cin 语句拆成以下两行代码：

```
cin >> numberOfBars
    >> oneWeight;
```

注意，和 cout 语句一样，每个 cin 只对应一个分号。

程序抵达 cin 语句时，它会等待用户从键盘输入。它将第一个变量设为从键盘输入的第一个值，第二个变量设为从键盘输入的第二个值，依此类推。但除非按 Enter 键，否则程序不会真正读取输入。利用这个设计，可在输入一行内容时按 Backspace 键纠错。

输入的各个数字必须以一个或多个空格或者以一个换行符来分隔。例如，假定要输入两个数字 12 和 5，但输入时没有用空格分隔它们，计算机就认为输入的是一个数字，即 125。使用 cin 语句时，计算机会跳过任意数量的空格或换行符，径直找到下一个输入值。因此，无论你用一个还是多个空格，甚至用一个换行符来分隔，都是无关紧要的。

cin 语句

cin 语句将变量设为从键盘键入的值。

语法

```
cin >> Variable 1 >> Variable 2 >>…;
```

示例

```
cin >> number >> size;
cin >> timeToGo
    >> pointsNeeded;
```

设计输入和输出

输入和输出通常称为 I/O，是程序中用户看得见的部分。I/O 设计不佳，就无法取悦于用户。

计算机执行 cin 语句时，它希望收到从键盘输入的数据。用户不输入数据，计算机会一直等下去。程序必须提示用户什么时候应该输入数字(或其他数据项)。计算机不会自动要求输入，所以示范程序包含了以下输出语句：

```
cout << "Enter the number of candy bars in a package\n";
cout << "and the weight in ounces of one candy bar.\n";
cout << "Then press return.\n";
```

这些输出语句**提示**用户输入。程序希望获取输入时一定要明确提示。

从终端输入数据时，输入的内容会在屏幕光标位置出现。虽然如此，始终应该在程序终止之前，主动显示输入的值。这称为**回显输入**，作用是让用户检查输入是否被正确读取。之所以要回显，是因为有时在屏幕上看起来没有问题的输入不一定能被计算机正确读取。例如，其中可能含有被忽视的打字错误或其他问题。通过回显输入，能对输入数据的完整性进行测试。

编程提示：I/O 中的换行

可在同一行输出和输入，而且对用户而言，这有时能产生更友好的界面。如果在最后一个提示行末尾省略\n 或 endl，用户的输入就会在这个提示行上出现。例如，假设使用以下提示和输入语句：

```
cout << "Enter the cost per person: $";
cin >> costPerPerson;
```

程序执行 cout 语句时，屏幕上将出现以下提示：

```
Enter the cost per person: $
```

用户输入的数据将在同一行上出现，如下所示：

```
Enter the cost per person: $1.25
```

■

√ 自测题

8. 写一个输出语句在屏幕上生成以下消息①：

```
The answer to the question of
Life, the Universe, and Everything is 42.
```

9. 写一个输入语句，使用从键盘输入的值来填充 int 类型的变量 theNumber。在输入语句前添加一个提示语句，提示用户输入一个整数。

---

① “那个伟大问题的答案……关于生命、宇宙以及一切……是……42。”这句话来自道格拉斯•亚当斯的科幻名著《银河系漫游指南》。——译注

10. 输出 double 类型的数字时，在程序中添加什么语句能确保这种类型的值以正常格式输出，并具有 3 个小数位。

11. 写一个完整的 C++程序，要求在屏幕上显示 Hello world。不要求该程序具备其他功能。

12. 写一个完整的 C++程序，要求读入两个整数，并输出两个整数之和。一定要提示输入，回显输入，并清楚说明输出的是什么。

13. 写一个输出语句来输出一个换行符和一个制表符。

14. 写一个小程序，声明 double 类型的变量 one，two，three，four 和 five，分别初始化为 1.000，1.414，1.732，2.000 和 2.236。写一些输出语句来生成以下表格。使用制表符转义序列\t 来对齐不同的列。如果还不熟悉制表符，请在做这道题的时候试验它。制表符相当于打字机上的一个机械跳格位置，会导致输出从下一列(栏)开始，两列间距通常是 8 个空格的倍数。许多编辑器和大多数字处理程序都允许调整制表位。我们的输出不调整。

    输出应该如下所示：

```
N  Square Root
1  1.000
2  1.414
3  1.732
4  2.000
5  2.236
```

2.3　数据类型和表达式

他俩没戏。他不是她喜欢的类型。

——*鸡尾酒会上的闲言碎语*

int 类型和 double 类型

2 和 2.0 在概念上固然是同一个数字，但 C++将它们视为不同类型。2 是 int 类型，2.0 是 double 类型，因为后者包含了小数部分(即使小数部分为 0)。再次强调，计算机编程所涉及的数学概念与数学课上学到的略有区别。计算机的一些实际问题造成计算机中的数字有别于这些数字的抽象定义。C++中，整数的行为和你预期的一样。int 类型保存的就是整数。但 double 类型要麻烦一些。由于 double 类型只能存储有限位数的数字，所以计算机将 double 类型的值保存为近似值。相反，int 类型的值则作为精确值来保存。double 值的具体精度在不同计算机上是不同的。但无论如何，double 值的精度至少能达到 14 位。这个精度对于大多数应用程序已经足够。不过，即使在一些简单的情况下，它也有可能造成不易察觉的问题。因此，如果已知某个变量保存的值是整数，而且大小在计算机能接受的范围之内，就最好把它声明为 int。

double 数字常量在书写时有别于 int 数字常量。int 常量绝对不能包含小数点。double 常量则有两种写法。对于 double 常量，最简单的形式就是日常生活中的写法(带小数)。采用这种写法，double 常量必须包含小数点。double 和 int 常量有一个共同点：C++的任何数字都不能包含逗号。

double 类型的常量的一种更复杂的写法称为**科学记数法**或**浮点记数法**，特别适合表达

非常大的数和非常小的小数。例如，数字 3.67×10^{17} 相当于 367 000 000 000 000 000.0，在 C++中最好表示成常量 3.67e17。再如数字 5.89×10^{-6}，它相当于 0.000 005 89，在 C++中最好表示成常量 5.89e-6。e 代表 exponent(指数)。e-6 相当于“乘以 10 的-6 次方”。

之所以要使用 **e 记数法**，是因为通常无法直接通过键盘输入作为上标的指数。e 后面的数字指出了两点：第一，小数点的移动方向；第二，要移多少位。例如，要把 3.49e4 变成一个没有 e 的数字，需要让小数点右移 4 位，得到 34900.0，这是同一个数的另一种写法。如果 e 后面的数字为负，小数点就左移指定位数，必要时插入 0。所以，3.49e-2 相当于 0.0349。

e 之前的数字可包含小数点，但这并非必须。但 e 之后的指数绝对不能有小数点。

由于计算机内存有限，所以数字通常只能用数量有限的字节来存储(也就是说，只能占用有限的存储空间)。所以数字大小也是有限的，而且这一限制因数字的类型而异。double 类型的最大值肯定大于 int 类型的最大值。目前大多数 C++实现都规定 int 类型的值不得超过 2 147 483 647，而 double 类型的值不得超过 10^{308}。

double 缘起

为什么带小数的数是 double 类型？是否还有 single 类型，大小只有 double 的一半？答案是既错又对。过去，许多编程语言用两种类型表示带小数的数。一种类型使用内存较少，但不太精确(也就是说，它不允许非常多的有效位)。第二种类型占用的内存是第一种类型的两倍(double 由此而来)，所以精度更高。与此同时，它还支持更大的数(尽管程序员对精度的关心远胜于大小)。使用两倍内存的这类数字称为“双精度”数字，使用较少内存的则称为“单精度”数字。在 C++中，根据这一传统约定，与双精度类型对应的类型称为 double 类型，与单精度对应的则是 float 类型。C++还为带小数的数提供了第三种类型，即 long double 类型。这些类型将在下一小节描述。不过，本书没有任何场合需要使用 float 和 long double 类型。

其他数值类型

除了 int 和 double，C++还有其他数值类型，图 2.2 总结了其中一部分。

图 2.2　部分数值类型

类型名称	占用的内存	取值范围	精度
short(或 short int)	2 字节	-32 767～32 767	(不适用)
int	4 字节	-2 147 483 647～2 147 483 647	(不适用)
long(或 long int)	4 字节	-2 147 483 647～2 147 483 647	(不适用)
float	4 字节	约 10^{-38}～10^{38}	7 位
double	8 字节	约 10^{-308}～10^{308}	15 位
long double	10 字节	约 10^{-4932}～10^{4932}	19 位

<本表格仅供参考，目的是让你体会不同类型的区别。在具体每个人的系统上，情况可能不同。“精度”是指有效位数，其中包括小数点前的位。float，double 和 long double 类型显示的取值范围是正数范围。负数也有一个类似的范围，但要在每个数字之前添加负号>

不同数值类型允许不同取值范围，也允许或高或低的精度。图 2.2 给出的占用内存、取值范围和精度仅供参考，目的是让你体会各种类型的区别。这些指标因系统而异。在你的系统上，情况可能不同。

虽然有的类型要用两个单词来拼写，但在声明这些类型的变量时，与声明 int 及 double 类型的变量并无区别。例如，以下语句声明一个 long double 类型的变量：

```
long double bigNumber;
```

类型名称 long 和 long int 代表同一个类型。所以，以下两个声明是等价的：

```
long bigTotal;
long int bigTotal;
```

当然，程序只能在上述两个声明中选择一个来声明变量 bigTotal。但具体选择哪个无关紧要。还要记住，类型名称 long 等价于 long int，而不是等价于 long double。

用于表示整数的类型(比如 int 和其他类似的类型)统称为**整数类型**或者**整型**。用于表示带小数点的数字的类型(比如 double 和其他类似类型)则统称为**浮点类型**或者**浮点型**。之所以称为“浮点”，是因为计算机在存储像 392.123 这样正常写法的数字时，首先会将数字转换成与 e 记数法类似的一种形式(本例是 3.92123e2)。计算机执行这种转换时，小数点会“浮动”(也就是移动)到一个新位置。

虽然 C++支持其他数值类型，但本书只用到了 int 和 double，偶尔用一下 long。对于大多数简单应用程序，除了 int 和 double，其他任何类型都派不上用场。需要大整数的程序可考虑 long。

C++11 类型

整型的大小在不同计算机上可能不一样。例如，32 位机器的整数可能是 4 字节，而 64 位机器可能是 8 字节。如需确定一个整型的取值范围，这就可能成为问题。C++11 增加了新整型来解决问题。新整型指定了确切大小以及是否有符号。这些类型通过包含 <cstdint>来使用。图 2.3 列出了部分类型。本书主要使用传统 int 和 long。但如果要指定确切大小，请考虑使用 C++11 类型。

图 2.3　部分 C++11 定宽整型

类型名称	占用的内存	取值范围
int8_t	1 字节	−128～127
uint8_t	1 字节	0～255
int16_t	2 字节	−32 768～32 767
uint16_t	2 字节	0～65 535
int32_t	4 字节	−2 147 483 648～2 147 483 647
uint32_t	4 字节	0~4 294 967 295
int64_t	8 字节	−9 223 372 036 854 775 808～9 223 372 036 854 775 807
uint64_t	8 字节	0～18 446 744 073 709 551 615
long long	至少 8 字节	

C++11 还提供 auto 类型，能根据等号右侧的表达式推断变量类型。例如，以下代码定义名为 x 的变量，它的数据类型由 expression 的求值结果决定。

```
auto x = expression;
```

该功能目前效果不彰，但以后定义自己的数据类型时可利用它简化代码。

C++11 还增加了判断变量或表达式类型的功能。decltype (expr)是变量或表达式 expr 的已声明类型，可在声明新变量时使用。例如：

```
int x = 10;
decltype (x*3.5) y;
```

上述代码声明 y 的类型和 x*3.5 一样。表达式 x*3.5 的结果是 double，所以 y 被声明为 double。

视频讲解：*C++11 Fixed Width Integer Types*

char 类型

我不想让你觉得计算机和 C++只会执行数值计算。接下来要介绍一种非数值类型。以后还会遇到其他更复杂的非数值类型。char(character 的缩写)类型的值表示单独一个符号，比如字母、数字或标点符号。在书籍和日常对话中，经常将这种类型的值称为**字符**。在 C++程序中，这种类型必须拼写成缩写形式 char。例如，char 类型的变量 symbol 和 letter 可以像下面这样声明：

```
char symbol, letter;
```

char 类型的变量可容纳键盘上的任何单独字符。所以，变量 symbol 可容纳一个'A'、一个'+'或者一个'a'。注意，字母大写和小写形式被视为不同字符。

使用 cout 输出的双引号中的文本则称为**字符串**值。例如，下面就是一个字符串，它来自图 2.1 的程序：

```
"Enter the number of candy bars in a package\n"
```

注意，字符串常量放到双引号内，而 char 类型的常量放到单引号内。两种引号含义不同。例如，'A'和"A"不同。'A'是 char 值，可存储到 char 类型的变量中。而"A"是字符串值。虽然这个字符串恰好只包含一个字符，但这一事实并不能使"A"成为 char。还要注意，无论字符串还是字符，左右两个引号是一样的(都是"或')。

图 2.4 演示如何使用 char 类型。注意在第一个和第二个姓名首字母之间输入了一个空格。但程序会跳过空格，将字母 B 作为输入的第二个字母来读取。通过 cin 将输入读入 char 类型的变量时，计算机会忽略所有空白间隔和换行符，直到遇到第一个非空白字符，并将其读入变量。输入中有无空白字符(比如空格、制表符等)对计算机没有区别。运行图 2.4 的程序，无论在两个姓名首字母之间插入一个空格(如示范对话所示)，还是连续输入两个首字母(不插入空格，即直接输入 JB)，结果都是一样的。

图 2.4　char 类型

```
1  #include <iostream>
2  using namespace std;
```

```
 3  int main()
 4  {
 5      char symbol1, symbol2, symbol3;

 6      cout << "Enter two initials, without any periods:\n"; // 输入两个姓名首字母，不要加点号
 7      cin >> symbol1 >> symbol2;
 8      cout << "The two initials are:\n";
 9      cout << symbol1 << symbol2 << endl;
10      cout << "Once more with a space:\n";
11      symbol3 = ' ';
12      cout << symbol1 << symbol3 << symbol2 << endl;
13      cout << "That's all.";
14      return 0;
15  }
```

示范对话

```
Enter two initials, without any periods:
J B
The two initials are:
JB
Once more with a space:
J B
That's all.
```

bool 类型

下个类型是 bool。该类型由 ISO/ANSI(国际标准化组织/美国国家标准化组织)于 1998 年增补到 C++语言中。bool 类型的表达式称为布尔(Boolean)表达式，这一名称源于英国数学家 George Boole(1815－1864)，他是数学逻辑规则的奠基人。

布尔表达式的求值结果只有两个：true 或 false。布尔表达式在分支和循环语句中使用。2.4 节更详细地讨论布尔表达式和 bool 类型。

string 类简介

虽然 C++缺乏原生数据类型来直接操作字符串，但在 string 类的帮助下，可采取和原生数据类型相似的方式处理字符串。类和原生数据类型的区别在第 10 章讨论。第 8 章更详细地讨论 string 类。

使用 string 类必须先包含 string 库：

```
#include <string>
```

程序还必须包含以下代码，通常放到文件起始处：

```
using namespace std;
```

声明 string 类型的变量和声明 int 或 double 类型的变量一样。例如，以下代码声明 string 类型的一个变量，并在变量中保存单词"Monday"：

```
string day;
day = "Monday";
```

如图 2.5 所示，可用 cin 将数据读入字符串。在两个字符串之间使用“+”符号可将两个字符串连接成一个更长的。例如以下代码：

```
string day, day1, day2;
day1 = "Monday";
```

```
day2 = "Tuesday";
day = day1 + day2;
```

会连接成以下字符串：

```
"MondayTuesday"
```

注意，连接两个字符串不会自动添加空格。这样的空格只能显式添加，如下所示：

```
day1 + " " + day2
```

用 cin 将输入的内容读入 string 变量时，除非遇到空白字符，否则计算机会一直读下去。空白字符是指在屏幕上所有显示为空白的字符，包括空格、制表符和换行符'\n'。这意味着不能输入含空格的字符串，否则就会出错。图 2.5 的示范对话 2 演示了一个例子。在本例中，用户试图输入"Mr. Bojangles"作为宠物名，但字符串只能读到"Mr."为止，因为下个字符是空格。"Bojangles"字符串被程序忽略，但假如有另一个 cin 语句，那么接下来就会读入它。第 8 章将介绍输入含空格字符串的一个技术。

图 2.5　string 类

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4  int main()
5  {
6      string middleName, petName;
7      string alterEgoName;
8      // ego 是拉丁文“我”的意思，alter ego 就是“另一个我”，或者说“知己”
9      cout << "Enter your middle name and the name of your pet.\n";
10     cin >> middleName;
11     cin >> petName;
12
13     alterEgoName = petName + " " + middleName;
14
15     cout << "The name of your alter ego is ";
16     cout << alterEgoName << "." << endl;
17
18     return 0;
19 }
```

示范对话 1

```
Enter your middle name and the name of your pet.
Parker Pippen
The name of your alter ego is Pippen Parker.
```

示范对话 2

```
Enter your middle name and the name of your pet.
Parker
Mr. Bojangles
The name of your alter ego is Mr. Parker.
```

类型的兼容性

一种类型的值一般不能存入另一种类型的变量。例如，大多数编译器对以下语句报错：

```
int intVariable;
intVariable = 2.99;
```

这是因为类型不匹配。常量 2.99 是 double 类型，变量 intVariable 是 int 类型。遗憾的是，并非所有编译器都以相同的方式响应上述赋值语句。有的显示错误消息，有的显示警告消息，有的则完全无动于衷。但即使编译器允许这种赋值语句，赋给变量 intVariable 的值也可能是 2，而非四舍五入后的 3。由于无法保证编译器允许这种赋值语句，所以要避免直接将 double 值赋给 int 变量。

常量 2.99 替换成 double 类型的变量问题依旧。大多数编译器对以下赋值语句报错：

```
int intVariable;
double doubleVariable;
doubleVariable = 2.00;
intVariable = doubleVariable;
```

虽然 2.00 不需要四舍五入，但问题依旧存在。2.00 是 double 而不是 int。如后文所述，可将 2.00 替换为 2 赋给 doubleVariable，但这依旧无法让编译器正常赋值。变量 intVariable 和 doubleVariable 的类型不同，这才是关键。

即使编译器允许在赋值语句中混用不同类型，但大多数情况下都不应该这样做。这不利于程序的移植，而且会引起混淆。例如，如果编译器允许将 2.99 赋给 int 变量，该变量获得的将是 2 而非 2.99，这便产生了混淆，因为程序的意思是把 2.99 赋给变量。

一些特殊情况允许将一种类型的值赋给另一种类型的变量。将 int 类型的值赋给 double 类型的变量是允许的。例如，以下语句既有效也合乎情理：

```
double doubleVariable;
doubleVariable = 2;
```

以上语句将 doubleVariable 变量的值设为 2.0。

虽然不建议，但确实可以将 int 值(如 65)保存到 char 类型的变量中。另外，还可将字母(如'Z')保存到 int 类型的变量中。从多方面考虑，C 语言将字符视为小整数值。遗憾的是，C++从 C 沿袭了这一点。之所以选择这样做，是因为 char 类型的变量占用内存比 int 类型少。所以，用 char 类型的变量进行计算可节省一些内存。不过，更清晰的方案是在操作整数时使用 int 类型，操作字符时使用 char 类型。

一般的规则是，不要将一种类型的值放到另一种类型的变量中——虽然违反这个规则的情况似乎比遵守这个规则的情况多。即使编译器没有严格贯彻这一规则，你也应该主动遵守。将一种类型的数据放入另一种类型的变量中可能造成问题。因为值必须转换成相应类型的值，但转换后的值可能不是你想要的。

bool 类型的值可赋给整型(short，int 和 long)变量，整数也可赋给 bool 变量。但这样做不好，应尽量避免。考虑到知识的全面性，并帮助你阅读其他人写的代码，我打算说具体点：赋给 bool 类型的变量时，任何非零的整数都将保存为 bool 值 true，值 0 则保存为 bool 值 false。反之，将 bool 值赋给整数变量时，true 作为 1 存储，false 则作为 0 存储。

算术操作符和表达式

C++程序对象程序利用算术操作符合并变量和/或数字。这些操作符包括+(加)、-(减)、*(乘)和/(除)。例如，图 2.1 的程序用操作符*求两个变量值的积，结果放入等号左侧的

变量：

```
totalWeight = oneWeight * numberOfBars;
```

所有算术操作符都可以作用于两个 int 或两个 double，甚至可以一样一个。但确切的结果值及其类型取决于被合并的两个数字的类型。两个操作数(也就是两个数字)都是 int，用算术操作符合并的结果也是 int。一个(或两个)操作数是 double，结果就会变成 double。例如，假定变量 baseAmount 和 increase 都是 int，则以下表达式的结果也是 int：

```
baseAmount + increase
```

但其中任何一个是 double，结果就是 double。将+换为-，*或/，结论一样。

结果类型的重要性可能超乎你的想象。例如，7.0/2 有一个 double 类型的操作数，即 7.0，所以结果是 double 值 3.5。但 7/2 的两个操作数都是 int，所以结果是 int 值 3。即使结果表面上相等，但实际可能还是有区别。例如，6.0/2 有一个 double 操作数，即 6.0，所以结果是 double 值 3.0，它只是一个近似值。但 6/2 有两个 int 操作数，所以结果是 int 值 3，这是一个精确值。除法操作符是受参数类型影响最大的操作符。

使用除法操作符时，如果一个或两个操作数是 double 类型，结果是符合预期的。但为 int 类型的两个操作数使用除法操作符，只能获得除法运算结果的整数部分。也就是说，整数除法丢弃了小数部分。所以，10/3 的结果是 3(而不是 3.3333...)，5/2 的结果是 2(而不是 2.5)，11/3 的结果是 3 (而不是 3.6666...) 。注意这些数字没有四舍五入，小数部分被直接丢弃，不管它有多大。

将操作符%应用于 int 类型的操作数，可还原使用/对两个整数进行除法运算时丢失的信息。应用于 int 值时，两个操作符/和%将生成执行长除法时获得的两个数(长除法是中学知识)。例如，17 用 5 除，商 3 余 2。/运算得到商，%得到余。例如，以下语句：

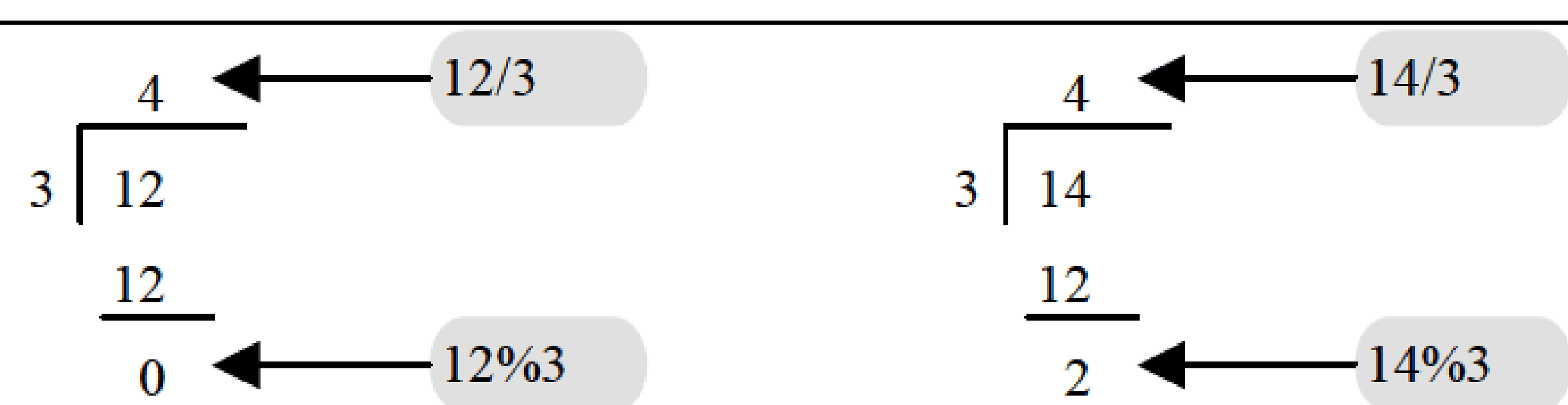
```
cout << "17 divided by 5 is " << (17/5) << endl;  // 商
cout << "with a remainder of " << (17%5) << endl;  // 余
```

输出如下：

```
17 divided by 5 is 3
with a remainder of 2
```

图 2.6 演示了将操作符/和%应用于 int 值时的情况。

图 2.6　整数除法



向负 int 值应用/和%操作符的结果因 C++实现而异。所以，为确保程序的可移植性，只有在确定两个 int 都非负时，才为它们使用操作符/和%。

算术表达式中可加入合理的空白间距。可在操作符和圆括号前后插入空格，当然也可以不加。只要易于阅读，加或不加可自由选择。

可插入圆括号来指定运算顺序，如以下两个表达式所示：

```
(x + y) * z
x + (y * z)
```

对第一个表达式求值，计算机首先使 x 和 y 相加，再将结果乘以 z。对第二个表达式求值，计算机首先使 y 和 z 相乘，再将结果加到 x 上。尽管数学公式可以使用方括号和其他形式的括号，但 C++不允许。C++只允许在算术表达式中使用圆括号来指定运算顺序。其他形式的括号有其他用途。

如省略圆括号，计算机将根据**优先级规则**决定操作符(如+和*)的求值顺序。这些优先级规则与代数和其他数学课采用的规则相似。例如，对以下表达式求值时：

```
x + y * z
```

将先乘后加。除了一些非常标准的情况(比如一连串的加法，或者在加法运算中嵌入简单乘法)，否则最好坚持用圆括号指定执行顺序。圆括号使表达式更易理解，并避免出错。附录 2 给出了完整的 C++优先级规则。

视频讲解：*Precedence and Arithmetic Operators*

图 2.7 列出了一些常见的算术表达式和对应的 C++表达式。

图 2.7　算术表达式

数学公式	C++表达式
$b^2 - 4ac$	`b*b - 4*a*c`
$x(y + z)$	`x * (y + z)`
$\frac{1}{x^2 + x + 3}$	`1 / (x*x + x + 3)`
$\frac{a + b}{c - d}$	`(a + b)/(c - d)`

陷阱：除法中的整数

两个整数相除结果也是整数。但如果期望的是小数就会成为问题。另外，该问题往往不易察觉，导致程序看起来正确，但会产生不正确的输出。例如，假定园林设计师设计公路景观时按每英里 5000 美元收费。已知以英尺为单位的公路长度，使用以下 C++语句能轻松计算出应收取费用：

```
totalPrice = 5000 * (feet/5280.0);
```

1 英里 5280 英尺，上述语句可行。假定公路长度 15000 英尺，以下公式获得总价格：

```
5000 * (15000/5280.0)
```

C++程序按以下顺序获得终值：15000/5280.0 结果是 2.84。然后 5000 和 2.84 相乘，得到 14200.00。借助于你写的 C++程序，可知该工程应收 14200 美元。

假定 feet 是 int 类型，同时忘记包含小数点和后面的 0，赋值语句变为以下形式：

```
totalPrice = 5000 * (feet/5280);
```

看起来没错，但会造成严重问题。使用这种形式的赋值语句，表示要使两个 int 类型

的值相除，所以 feet/5280 相当于 15000/5280，结果是 int 值 2(而不是你希望的 2.84)。所以，赋给 totalPrice 的值是 5000*2，或者说 10000。忘记小数点，收取的费用就变成 10000 美元。但如前所述，实际应收款是 14200 美元。遗漏小数点就蒙受了 4200 美元的损失。注意，totalPrice 的类型无论是 int 还是 double，都无法改变这一事实。损失发生在向 totalPrice 赋值之前。■

√ 自测题

15. 将以下数学公式转换为 C++表达式：

    $3x$　　$3x+y$　　$\frac{x+y}{7}$　　$\frac{3x+y}{z+2}$

16. 以下几行程序输出什么(假定它们嵌入一个正确的程序，而且所有变量都声明为 char 类型)：

```
a = 'b';
b = 'c';
c = a;
cout << a << b << c << 'c';
```

17. 以下两行程序输出什么(假定它们嵌入一个正确的程序，而且 number 声明为 int 类型)：

```
number = (1/3) * 3;
cout << "(1/3) * 3 is equal to " << number;
```

18. 写完整 C++程序将两个整数读入两个 int 变量，输出第一个数除以第二个数的商和余。可用操作符/和%来实现。

19. 给定以下代码段：

```
double c = 20;
double f;
f = (9/5) * c + 32.0;
```

    它的目的是将摄氏温度(c)换算为华氏温度(f)，请回答以下问题。
    a. 赋给 f 的值是什么？
    b. 解释实际发生的情况和程序员的本意。
    c. 重写代码使之符合程序员的本意。

20. 以下程序的输出结果是什么(假定它们嵌入一个正确的程序，已将 month, day, year 和 date 声明为 string 类型)？

```
month = "03";
day = "04";
year = "06";
date = month + day + year;
cout << date << endl;
```

更多赋值语句

赋值操作符(=)可以和算术操作符合并，使变量和值进行加、减、乘、除来改变自身。常规形式如下：

```
Variable Op= Expression
```

它等价于：

```
Variable = Variable Op (Expression)
```

其中，*Op* 是操作符(比如+，-和*)。*Expression* 可以是另一个变量、常量或者更复杂的表达式。下面给出一些例子。

示例	等价形式
`count += 2;`	`count = count + 2;`
`total -= discount;`	`total = total - discount;`
`bonus *= 2;`	`bonus = bonus * 2;`
`time /= rushFactor;`	`time = time / rushFactor;`
`change %= 100;`	`change = change % 100;`
`amount *= cnt1 + cnt2;`	`amount = amount * (cnt1 + cnt2);`

2.4　简单控制流程

"如果你以为我们是蜡做的人像，那你就应该先付钱，"他说，"你知道，蜡像不是做来给人白看的。嘿！不是的！"

"反过来说，"那个有着"弟"字的小胖子说，"如果你认为我们是活的，你就应该说话。"

——*刘易斯·卡洛尔，《爱丽丝镜中奇遇记》*

前面的程序都由一些简单的语句构成。语句按给定顺序逐行执行。但更复杂的程序要求能以某种方式改变语句执行顺序。语句执行顺序通常称为**控制流程**或**控制流**。本节介绍向程序添加控制流程的两种简单方式。要讨论一种分支机制，允许程序在两个备选行动中选一个，具体选择哪个由变量值决定。还要讨论一种循环机制，允许重复一个行动。

一个简单的分支机制

有时需要让程序根据输入从两个备选行动中选一个。以计算计时员工周薪为例，假定公司薪水制度是每周工作 40 小时以内，按正常工资计算。超出 40 小时算加班，按正常工资 1.5 倍计算。员工一周内的工作时间大于或等于 40 小时，工资计算公式如下：

```
rate * 40 + 1.5 * rate * (hours - 40)  // rate 是时薪
```

但员工工作时间也可能少于 40 小时。这种情况下上述公式会算出负值(要体会这一点，可将 10 代入 hours，将 1 代入 rate，并执行计算。倒霉的员工领到的薪水将为负数。如工作时间少于 40 小时，正确薪水计算公式其实是非常简单的，如下所示：

```
rate * hours
```

要同时支持这两种情况(工作时间大于或等于 40 小时，或小于 40 小时)，程序必须能在这两个公式之间做出选择。为计算员工的周薪，程序要采取的行动如下所示：

```
判断(hours > 40)是否为 true
如果是，执行以下赋值语句：
   grossPay = rate * 40 + 1.5 * rate * (hours - 40);
如果不是，执行以下赋值语句：
   grossPay = rate * hours;
```

有一个 C++语句专门执行这种分支行动。**if-else 语句**会在两个备选行动之间选择。例如，前面讨论的周薪计算可用以下 C++语句完成：

```
if (hours > 40)
    grossPay = rate * 40 + 1.5 * rate * (hours - 40);
else
    grossPay = rate * hours;
```

图 2.8 用一个完整程序演示了该语句的用法。

图 2.8　if-else 语句

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5      int hours;
6      double grossPay, rate;
7      cout << "Enter the hourly rate of pay: $";
8      cin >> rate;
9      cout << "Enter the number of hours worked,\n"
10          << "rounded to a whole number of hours: ";
11     cin >> hours;

12     if (hours > 40)
13         grossPay = rate*40 + 1.5*rate*(hours - 40);
14     else
15         grossPay = rate*hours;
16     cout.setf(ios::fixed);
17     cout.setf(ios::showpoint);
18     cout.precision(2);
19     cout << "Hours = " << hours << endl;
20     cout << "Hourly pay rate = $" << rate << endl;
21     cout << "Gross pay = $" << grossPay << endl;
22     return 0;
23 }
```

示范对话 1

```
Enter the hourly rate of pay: $20.00
Enter the number of hours worked,
rounded to a whole number of hours: 30
Hours = 30
Hourly pay rate = $20.00
Gross pay = $600.00
```

示范对话 2

```
Enter the hourly rate of pay: $10.00
Enter the number of hours worked,
rounded to a whole number of hours: 41
Hours = 41
Hourly pay rate = $10.00
Gross pay = $415.00
```

图 2.9 总结了 `if-else` 语句的两种形式。第一种是最简单的 `if-else` 语句，第二种是 2.4.4 节“复合语句”要讨论的主题。第一种形式，两个语句可为任何可执行语句。*Boolean_Expression* 是测试，求值结果要么为 `true`，要么为 `false`。检查该表达式的值就知道是否满足条件。之前的 `if-else` 语句的 *Boolean_Expression* 是：

```
hours > 40
```

程序到达 if-else 语句时，实际执行两个内嵌语句之一。*Boolean_Expression* 为 true(满足条件)，执行 *Yes_Statement*。*Boolean_Expression* 为 false(不满足条件)，执行 *No_Statement*。注意，*Boolean_Expression* 必须包含在圆括号内(这是 C++ if-else 语句的语法规则)。

图 2.9　if-else 语句的语法

每个备选行动只有一个语句：

```
1  if (Boolean_Expression)
2      Yes_Statement
3  else
4      No_Statement
```

每个备选行动有一组语句：

```
5  if (Boolean_Expression)
6  {
7      Yes_Statement_1
8      Yes_Statement_2
9          . . .
10     Yes_Statement_Last
11 }
12 else
13 {
14     No_Statement_1
15     No_Statement_1
16         . . .
17     No_Statement_Last
18 }
```

布尔表达式是求值结果为 true 或 false 的表达式。if-else 语句肯定包含一个 *Boolean_Expression*。最简单的 *Boolean_Expression* 由两个表达式(比如数字或变量)构成，它们使用图 2.10 总结的某个比较操作符进行比较。注意有的操作符由两个符号构成，例如==，!=，<=和>=。注意，必须用双等号==表示相等，用!=表示不等。在两个符号构成的操作符中，两个符号之间不能有空格。执行 if-else 语句时，首先对要比较的两个表达式进行求值，再用操作符进行比较。结果为 true 时，执行第一个语句，否则执行第二个。

图 2.10　比较操作符

数学符号	中文说法	C++表示法	C++示例	等价的数学表示
=	等于	==	x + 7 == 2*y	x + 7 = 2y
≠	不等于	!=	ans != 'n'	ans ≠ 'n'
<	小于	<	count < m + 3	count < m + 3
≤	小于或等于	<=	Time <= limit	time ≤ limit
>	大于	>	Time > limit	time > limit
≥	大于或等于	>=	age >= 21	age ≥ 21

可用 and(逻辑与)操作符(在 C++中表示为&&)合并两个比较。例如，假定 x 大于 2，而且小于 7，则以下布尔表达式为 true(即条件满足)：

```
(2 < x) && (x < 7)
```

用&&连接两个比较时，只有两个比较的结果都为 true(两个条件都满足)，整个表达式才为 true；否则整个表达式为 false。

“逻辑与”操作符&&

可用逻辑与(and)操作符&&将两个简单测试合并成一个更复杂的布尔表达式。

语法(针对使用&&的布尔表达式)

```
(Comparison_1) && (Comparison_2)
```

示例(包含在一个 if-else 语句内)

```
if ( (score > 0) && (score < 10) )
    cout << "score is between 0 and 10.\n";
else
    cout << "score is not between 0 and 10.\n";
```

如果 score 的值大于 0，同时小于 10，就执行第一个 cout 语句；否则执行第二个。

还可使用 or(逻辑或)操作符(在 C++中表示为||)来合并两个比较表达式。例如，假定 y 小于 0 或 y 大于 12，则以下表达式为 true：

```
(y < 0) || (y > 12)
```

用||连接两个比较时，只要其中任何一个为 true 或者两个都为 true(满足任何条件)，整个表达式就为 true；否则，整个表达式为 false。

“逻辑或”操作符||

可用逻辑或(or)操作符||将两个简单测试合并为一个更复杂的布尔表达式。

语法(针对使用||的布尔表达式)

```
(Comparison_1) || (Comparison_2)
```

示例(包含在一个 if-else 语句内)

```
if ( (x == 1) || (x == y) )
    cout << "x is 1 or x equals y.\n";
else
    cout << "x is neither 1 nor equal to y.\n";
```

如果 x 等于 1，或者 x 等于 y (或者同时满足这两个条件)，就执行第一个 cout 语句，否则执行第二个。

记住，在 if-else 语句中使用布尔表达式时，必须用圆括号将布尔表达式封闭起来。例如，在一个 if-else 语句中，如 if 的条件使用了操作符&&，并包含两个比较，就要像下面这样用圆括号封闭 if 之后的整个比较表达式：

```
if ( (temperature >= 95) && (humidity >= 90) )
    . . .
```

注意，总共有两层圆括号，外层的圆括号是必需的，但内层的圆括号(用于封闭不同的比较)则不是必需的。但由于能使语句结构更清楚，所以通常都会包括它们。

可用求反操作符!对任何布尔表达式求反。如希望对一个布尔表达式求反，可以把表达式放入一对圆括号中，再把操作符!放到它的前面。例如，!(x < y)表示“x 不小于 y”。

由于 if-else 语句中的布尔表达式必须包含在一对圆括号中，所以在 if-else 语句中使用求反表达式时，要再用一对圆括号封闭求反表达式。例如，if-else 语句可能这样开始：

```
if (!(x < y))
    ...
```

可以避免使用!操作符。例如，上述 if-else 语句可更换成以下版本，它是等价的，而且更容易理解：

```
if (x >= y)
    ...
```

本书很晚的时候才会真正用到!操作符，届时更详细讨论。

有时希望 if-else 语句中的一个备选行动不做任何事情。在 C++中，这可以通过省略 else 部分来实现。这种形式的语句称为 **if 语句**，以区别于 if-else 语句。例如，以下两个语句中的第一个语句就是 if 语句：

```
if (sales >= minimum)          // 如果实际销售额大于或等于公司规定的最小值，
    salary = salary + bonus; // 就把奖金加上
cout << "salary = $" << salary;
```

假如 sales 的值大于或等于 minimum 的值，就执行赋值语句，再执行后续的 cout 语句。但是，假如 sales 的值小于 minimum，就不执行缩进的赋值语句。在这种情况下，if 语句不会导致任何更改(也就是说，不会有奖金加到基本工资里)，程序直接执行 cout 语句。

陷阱：连续的不等式

不要在程序中使用如下所示的连续不等式：

```
if (x < z < y)              ← 不要这样做！
cout << "z is between x and y.";
```

在程序中使用以上语句，程序也许能通过编译并运行，但毫无疑问，它会产生错误的结果。我们将在了解 C++语言更多的细节之后，再讨论这个问题的根源。使用任何比较操作符(而非仅仅是<)来进行一连串的比较时，都会发生同样的问题。表示连续不等式的正确方式是使用“逻辑与”操作符&&，如下所示：

```
if ( (x < z ) && (z < y) )         ← 正确的形式
    cout << "z is between x and y.";
```

■

陷阱：该用==的时候用了=

视频讲解：*Common Bugs with = and ==*

遗憾的是，在 C++中，你认为正确的许多 C++语句实际上都会产生歧义。换言之，即使语句在语法上正确，程序能成功编译并运行，不报告任何错误，但结果仍然和希望的不符。你可能没有意识到一些语句写错了，所以可能导致很严重的问题。即使发现结果不正确，但绞尽脑汁也想不出症结在哪里。一个常见的错误是在本该使用==的时候使用了=。例如像下面这样开头的一个 if-else 语句：

```
if (x = 12)
```

```
  Do_Something
else
  Do_Something_Else
```

假定要测试 x 的值是否等于 12，所以本意是用==而不是=。你也许以为编译器能捕捉这个错误，因为对于以下表达式来说：

```
x = 12
```

它不是逻辑表达式，不能表示一个条件是否满足。相反，它是赋值语句，所以编译器理应报错。遗憾的是，现实并非如此。在 C++中，表达式 x = 12 和 x + 12 与 2 + 3 一样，是一个能返回值的表达式(或者说是具有值的表达式)。整个赋值表达式的值就是传给等号左侧变量的值。例如，x = 12 的值就是 12。从前面对布尔值兼容性的讨论可知，int 值可转换成 true 或 false。由于 12 是一个非 0 的值，所以会转换成 true。如果将 x = 12 用作 if 语句中的布尔表达式，这个布尔表达式将始终为 true，所以，程序始终会执行第一个分支(*Do_Something*)。

这个错误很难发现，因为它看起来是正确的！相反，假如将 12 放在比较表达式左侧(如下所示)：

```
if (12 == x)
  Do_Something
else
  Do_Something_Else
```

那么一旦错误地用操作符=来取代上面的操作符==，编译器就会报错。

记住，在操作符==中少写一个=是许多编译器都不能捕捉的常见错误，这个错误很难发现，而且几乎肯定不会产生你希望的结果。C++的许多可执行语句可作为几乎任何表达式使用，其中包括 if-else 语句的布尔表达式。在本该使用布尔表达式的地方使用了赋值语句，赋值语句会被解释为布尔表达式。但此时的“测试”结果肯定和你希望的不符。上述 if-else 语句乍一看是对的，也能正常编译和运行，但结果可能出乎预料。 ■

复合语句

经常都要求 if-else 语句的每个分支执行多个语句。为此，可将每个分支的语句封闭到一对花括号中，如图 2.9 第二个语法模板所示，图 2.11 则给出了一个实际的例子。花括号内的一组语句统称为**复合语句**。在 C++中，复合语句被视为单个语句。凡是能使用单个语句的地方都能换成复合语句(因此，图 2.9 的第二个语法模板实际是第一个模板的特例)。图 2.11 包含两个复合语句，它们嵌入一个 if-else 语句中。

图 2.11　在 if-else 语句中使用复合语句

```
 1  if (myScore > yourScore)
 2  {
 3      cout << "I win!\n";
 4      wager = wager + 100;
 5  }
 6  else
 7  {
 8      cout << "I wish these were golf scores.\n";
 9      wager = 0;
10  }
```

if-else 的语法规则要求 *Yes_Statement* 和 *No_Statement* 分别只能是一个语句。一个分支要执行若干个语句，就必须将它们放到花括号中，转换成复合语句。编译器发现 if 和 else 之间有两个或多个不在花括号中的语句会报错。

√ 自测题

21. 写一个 if-else 语句，使其在 score 变量的值大于 100 时输出 High，在 score 的值小于等于 100 时输出 Low。score 变量是 int 类型。

22. 假设 savings (存款) 和 expenses (开销) 是 double 类型的变量，而且已经赋值。写一个 if-else 语句，如果 savings 大于等于 expenses，就使 savings 的值减去 expenses 的值，结果再赋给 savings。然后，将 expenses 的值设为 0，并输出单词 Solvent(有偿还能力)。但是，如果 savings 小于 expenses，if-else 语句就只输出单词 Bankrupt(破产)，不更改任何变量的值。

23. 写一个 if-else 语句，使其在变量 exam(考试成绩)的值大于或等于 60，而且变量 programsDone 的值大于或等于 10 的条件下，输出单词 Passed(通过)；否则，if-else 语句输出单词 Failed(没通过)。变量 exam 和 programsDone 都是 int 类型。

24. 写一个 if-else 语句，使其在变量 temperature(温度)大于或等于 100，或者变量 pressure(气压)的值大于或等于 200，或者同时满足这两个条件的前提下，输出单词 Warning(报警)；否则，if-else 语句输出单词 OK(正常)。变量 temperature 和 pressure 都是 int 类型。

25. 假设有以下二次方程式：

$$x^2 - x - 2$$

描述它在什么情况下为正(也就是大于 0)。换言之，需要描述要么小于较小根(-1)，要么大于较大根(+2)的一系列数字。写一个 C++布尔表达式，使其在方程式为正时求值为 true。

26. 假设有以下二次方程式：

$$x^2 - 4x + 3$$

描述它在什么情况下为负。换言之，需要描述不仅大于较小根(+1)，还要小于较大根(+3)的一系列数字。写一个 C++布尔表达式，使其在方程式为负时求值为 true。

27. 在 if-else 语句中嵌入的以下 cout 语句将输出什么？假设已将这些代码嵌入一个完整、正确的程序中。请解释你的答案。

a.
```
if (0)
  cout << "0 is true";
else
   cout << "0 is false";
cout << endl;
```

b.
```
if (1)
   cout << "1 is true";
else
   cout << "1 is false";
cout << endl;
```

c.
```
if (-1)
     cout << "-1 is true";
else
     cout << "-1 is false";
cout << endl;
```

注意：仅供练习，不应该遵循这样的编程风格。

简单的循环机制

大多数程序都包含需要多次重复的行动。以图 2.8 用于计算员工工资总额的程序为例。如果公司有 100 名员工，一个更完善的工资程序将重复这一计算 100 次。程序中，用于重复一个或一组语句的那部分称为**循环**。C++语言提供了多种方式创建循环。其中一个称为 **while 语句**，或者称为 **while 循环**。为了展示它的用法，首先介绍一个简单的练习程序，再介绍一个更实用的程序。

图 2.12 的程序含有简单的 while 语句(以粗体显示)。花括号{和}之间的部分称为 while 循环**主体**，也就是需要重复采取的行动。花括号内的语句依次执行，执行完一次之后，再从头继续执行，如此重复，直到 while 循环结束。在第一个示范对话中，循环主体在循环结束之前，重复执行了三次，所以程序输出了三次 Hello。循环主体的每一次重复都称为循环的一次**迭代**。因此，第一个示范对话表明循环进行了三次迭代。

图 2.12　while 循环

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5      int countDown;

6      cout << "How many greetings do you want? ";
7      cin >> countDown;

8      while (countDown > 0)
9      {
10         cout << "Hello ";
11         countDown = countDown - 1;
12     }

13     cout << endl;
14     cout << "That's all!\n";

15     return 0;
16 }
```

示范对话 1

```
How many greetings do you want? 3
Hello Hello Hello
That's all!
```

示范对话 2

```
How many greetings do you want? 1
Hello
That's all!
```

示范对话 3

```
How many greetings do you want? 0

That's all!
```

← 循环主体执行 0 次

“while” (当……时)一词已暗示了 while 语句的含义。当圆括号内的布尔表达式满足条件时(求值结果为 true)，就重复执行循环。在图 2.12 中，这意味着只要变量 countDown 大于 0，就重复执行循环主体。下面以第一个示范对话为例，研究一下 while 循环是如何执行的。用户输入 3，所以 cin 语句将 countDown 的值设为 3。在这种情况下，当程序到

达 while 语句时，肯定满足 countDown 大于 0 这个条件，所以执行循环主体中的语句。每次重复循环主体时，都会执行以下两个语句：

```
cout << "Hello ";
countDown = countDown - 1;
```

因此，循环主体每次重复都会输出"Hello"，同时变量 countDown 的值递减 1。计算机重复了三次循环主体后，变量 countDown 的值递减至 0，圆括号中的布尔表达式不再满足。所以，重复了三次循环主体后，这个 while 语句就会终止。

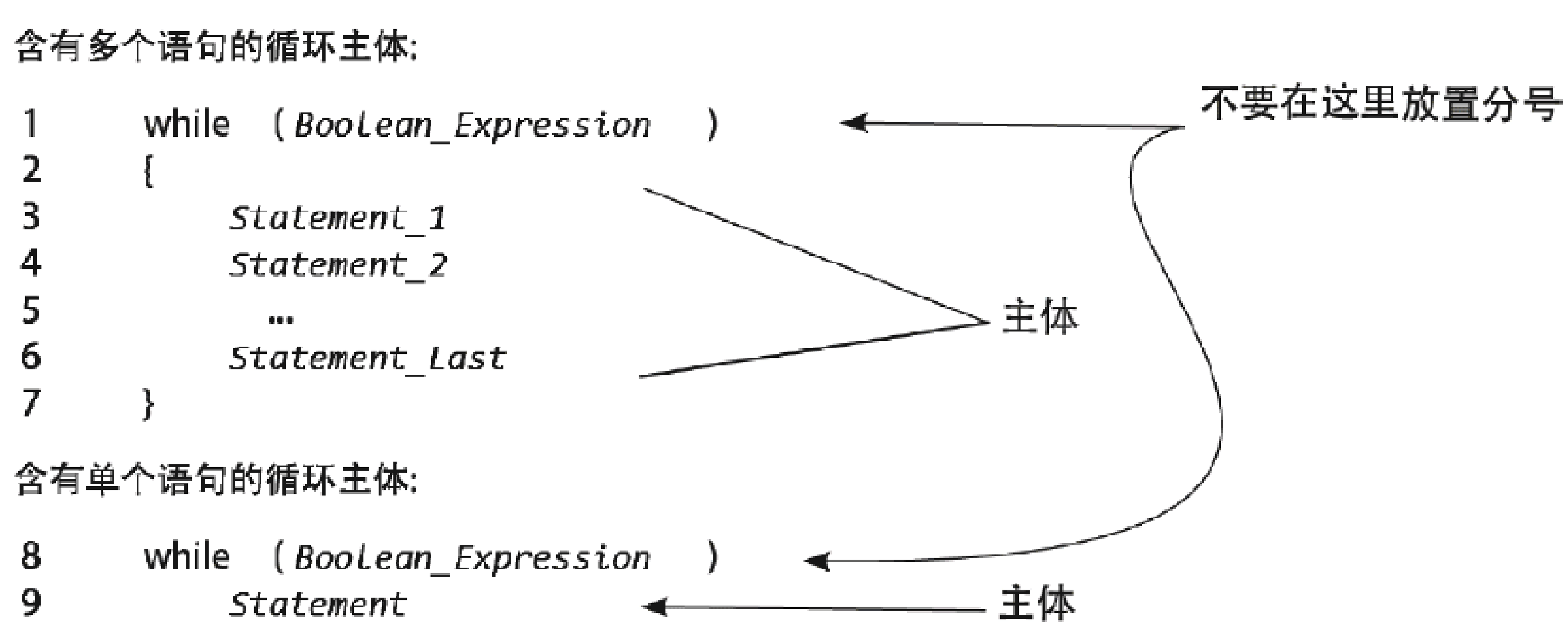
图 2.13 总结了 while 语句的语法。对 *Boolean_Expression* 的要求和对 if-else 语句中的布尔表达式的要求完全一样。和 if-else 语句一样，while 语句中的布尔表达式也必须用圆括号括起来。图 2.13 针对两种情况给出了语法模板：第一种情况是循环主体有多个语句。第二种情况是循环主体只有一个语句。注意，只有一个语句时不必使用花括号。

现在着重讨论 while 语句采取的行动。执行 while 语句时，发生的第一件事情是检查单词 while 之后的布尔表达式。该表达式的求值结果要么为 true，要么为 false。例如：

```
countDown > 0
```

如果 countDown 的值为正，则求值结果为 true；如果为 false，就不采取行动，程序将继续执行 while 语句之后的下一个语句。如果比较结果为 true，就执行整个循环主体。在被比较的表达式中，通常至少要有一个包含了即将由循环主体改变的东西，比如图 2.12 的 while 语句中的 countDown 的值。执行循环主体之后，会再次进行比较。只要比较结果为 true，这个过程就会不断地重复。循环主体每次迭代后，会再次进行比较，假如结果为 true，就再次执行整个循环主体。一旦比较结果不为 true，就结束 while 循环。

图 2.13　while 语句的语法



while 语句执行时，第一件事是检查布尔表达式。表达式不为 true，就永远不执行循环主体，具体可参见图 2.12 的示范对话 3。许多时候都要求循环主体执行 0 次。例如，假定 while 循环要读取所有不及格分数，但实际没人不及格，就应该让循环主体执行 0 次。

一个 while 循环可能执行 0 次循环主体，这是很正常的情况。但是，如果要求循环主体在任何情况下都至少执行一次，就应该使用 do-while 循环。do-while 语句与 while 语句相似，只是它的循环主体至少执行一次。do-while 语句的语法请参见图 2.14。

图 2.15 是使用了 do-while 循环的示范程序。do-while 循环的第一件事情就是执行循环主体语句。循环主体第一次迭代之后，do-while 的行为就和 while 一样了。换言之，

就是检查布尔表达式，结果为 true 就再次执行循环主体，再次检查布尔表达式，如此反复。

图 2.14　do-while 语句的语法

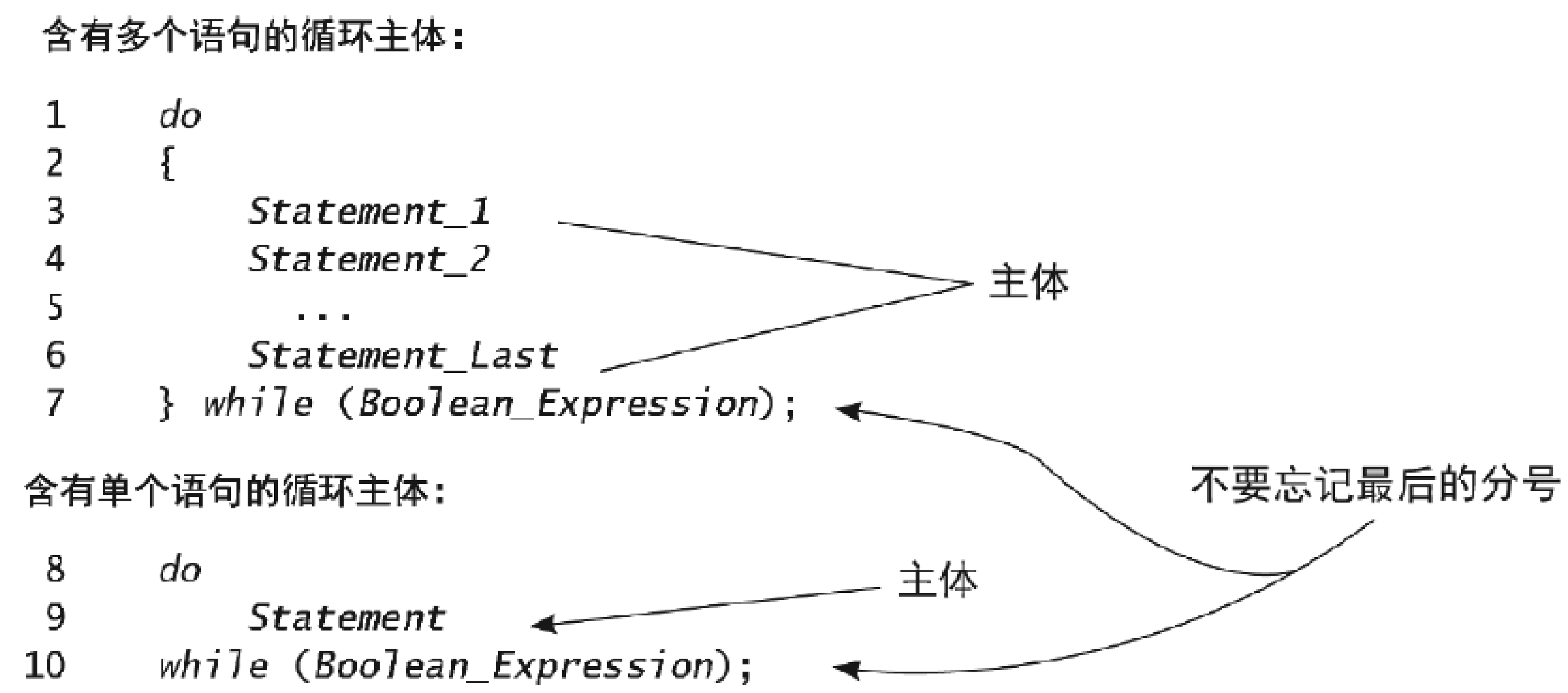


图 2.15　do-while 循环

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5      char ans;
6      do
7      {
8          cout << "Hello\n";
9          cout << "Do you want another greeting?\n"
10              << "Press y for yes, n for no,\n"
11              << "and then press return: ";
12         cin  >> ans;
13     } while (ans == 'y' || ans == 'Y');
14     cout << "Good-Bye\n";
15     return 0;
16 }
```

示范对话

```
Hello
Do you want another greeting?
Press y for yes, n for no, and then press return: y
Hello
Do you want another greeting?
Press y for yes, n for no, and then press return: Y
Hello
Do you want another greeting?
Press y for yes, n for no, and then press return: n
Good-Bye
```

递增操作符和递减操作符

2.3 节讨论了二元操作符。**二元操作符**有两个操作数。**一元操作符**则只有一个。之前用过两个一元操作符，即+和–，它们在+7 和–7 这样的表达式中使用。C++语言还有另外两个很常用的一元操作符，即++和--。操作符++称为**递增操作符**，操作符--称为**递减操作符**。它们通常与 int 类型的变量一起使用。假定 n 是 int 类型的变量，n++使 n 递增 1，n--使 n 递减 1。所以，n++和 n--(后跟一个分号)是可执行语句。例如以下语句：

```
int n = 1, m = 7;
n++;
cout << "The value of n is changed to " << n << endl;
m--;
cout << "The value of m is changed to " << m << endl;
```

产生的输出如下：

```
The value of n is changed to 2
The value of m is changed to 6
```

这时你想必会恍然大悟，猜到 C++这个名称中的“++”是怎么来的。

递增和递减语句常在循环中使用。例如，图 2.12 的 while 循环中使用了以下语句：

```
countDown = countDown - 1;
```

但大多数有经验的 C++程序员都会选择使用递减操作符。所以整个 while 循环可这样修改：

```
while (countDown > 0)
{
    cout << "Hello ";
    countDown--;
}
```

编程实例 信用卡余额

假定有一张信用卡，卡上已产生应还金额 50 美元，银行按 2%月利率收费。假定一直不还款，多少个月之后，这张卡的应还金额会超过 100 美元？解决这个问题的一个办法是查看每月账单，统计在应还金额达到或超过 100 美元之前，总共会经历多少个月。但更好的办法是用程序计算每月应还金额，而不必等着银行寄账单。通过这种方式，不需要漫长的等待(也不会影响自己的信用评级)，就能迅速得到答案。

一个月后，卡的应还金额是 50 美元加 50 美元的 2%，也就是 51 美元。两个月后，卡的应还金额是 51 美元加 51 美元的 2%，也就是 52.02 美元。三个月之后，卡的应还金额是 52.02 美元加 52.02 美元的 2%，依此类推。总之，每月应还金额都会增加 2%。该程序将应还金额保存到名为 balance 的变量中对其进行跟踪。每月对 balance 变量值的修改可以像下面这样进行：

```
balance = balance + 0.02 * balance;
```

重复这个行动，直到 balance 的值达到(或超过)100 美元，并对重复次数进行计数，就知道多少月后应还金额达到 100 美元。为此，需要用另一个变量对 balance 的修改次数进行计数。假定新变量名称是 count。while 循环中最终的主体将包含以下语句：

```
balance = balance + 0.02 * balance;
count++;
```

为了使循环正确执行，必须在循环执行之前将恰当的值赋给变量 balance 和 count。本例在声明变量的同时初始化。完整程序参见图 2.16。

图 2.16　信用卡程序

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5      double balance = 50.00;
6      int count = 0;
7      cout << "This program tells you how long it takes\n"
8           << "to accumulate a debt of $100, starting with\n"
9           << "an initial balance of $50 owed.\n"
10          << "The interest rate is 2% per month.\n";
11     while (balance < 100.00)
12     {
13         balance = balance + 0.02 * balance;
14         count++;
15     }
16     cout << "After " << count << " months,\n";
17     cout.setf(ios::fixed);
18     cout.setf(ios::showpoint);
19     cout.precision(2);
20     cout << "your balance due will be $" << balance << endl;
21     return 0;
22 }
23
```

示范对话

```
This program tells you how long it takes
to accumulate a debt of $100, starting with
an initial balance of $50 owed.
The interest rate is 2% per month.
After 36 months,
your balance due will be $101.99
```

陷阱：无限循环

只要 while 之后的布尔表达式为 true，while 或 do-while 循环就不会终止。布尔表达式通常包含一个将由循环主体更改的变量。变量应不断更改，最终使布尔表达式求值为 false，从而终止循环。但假如不小心让布尔表达式始终为 true，循环就会一直运行。这种循环称为**无限循环**。

首先描述一个能终止的循环。以下 C++代码输出小于 12 的所有正偶数。也就是说，它将逐行输出数字 2，4，6，8 和 10，然后循环终止：

```
x = 2;
while (x != 12)
{
    cout << x << endl;
    x = x + 2;
}
```

每次循环迭代，x 的值都递增 2，直到变成 12。此时，单词 while 之后的布尔表达式不再为 true，循环结束。

现在假设要输出小于 12 的奇数，而不是偶数。你可能以为只需这样修改初始化语句：

```
x = 1;
```

但这是错误的，它会导致无限循环。因为 x 的值会从 11 变成 13，永远不等于 12，所以循环永远不会终止。

用==或!=检查一个数来终止循环，很容易造成无限循环问题。处理数字时，更安全的做法始终是测试它是否超过了一个值。例如，将前面 while 循环的第一行改为以下语句，就可将 x 初始化为任意数字，循环始终都能正常终止：

```
while (x < 12)
```

无限循环的程序不能自动停止，除非强行终止。初学编程的读者可能写出含有无限循环的程序，所以最好学会怎样强行终止程序。强行终止程序的方法因系统而异。在很多系统上，按 Control+C(按 Ctrl 键的同时按 C 键)就可以终止程序。 ■

√ 自测题

28. 以下代码的输出是什么(假定它们已嵌入一个正确的程序，而且 x 已声明为 int 类型)?

```
x = 10;
while (x > 0)
{
    cout << x << endl;
    x = x - 3;
}
```

29. 在上一题中，将>替换为<会输出什么?

30. 以下代码的输出是什么(假定它们已嵌入一个正确的程序，而且 x 已声明为 int 类型)?

```
x = 10;
do
{
    cout << x << endl;
    x = x - 3;
} while (x > 0);
```

31. 以下代码的输出是什么(假定它们已嵌入一个正确的程序，而且 x 已声明为 int 类型)?

```
x = -42;
do
{
    cout << x << endl;
    x = x - 3;
} while (x > 0);
```

32. while 语句和 do-while 语句有何重要区别?

33. 以下代码的输出是什么(假定它们已嵌入一个正确的程序，而且 x 已声明为 int 类型)?

```
x =10;
while (x > 0)
{
    cout << x << endl;
    x = x + 3;
}
```

34. 写完整 C++程序逐行输出 1～20 的整数。该程序不执行别的任务。

2.5 程 序 风 格

在非常重要的事情上，最要紧的并非真诚，而是格调 ①。

——*奥斯卡·王尔德，《不可儿戏》*

我们的示范程序中，所有变量名都准确表达了它们的用途。示范程序在编排上采用了一种特定的格式。例如，声明和语句的缩进量是相同的。这些风格不仅仅是为了美观。风格好的程序更容易阅读，更容易纠正，也更容易修改。

缩进

程序应合理编排，使原本就是一个整体的各元素能组合到一起。一种方法是在逻辑上独立的各个部分之间插入空行。合理的缩进也有助于澄清程序结构。包含在一个语句内部的其他语句应该缩进。特别是，if-else 语句、while 循环和 do-while 循环应该缩进。缩进可采用示范程序的方式，也可采用其他类似的方式。

花括号{}用于确定程序结构中的一个较大的部分。像示范程序中那样，使每个花括号都单独占一行，以便定位一对匹配的花括号。注意，我们有意缩进了一些花括号。如果一对花括号要嵌入另一对中，嵌入的那一对花括号的缩进量要大于外面那一对，具体可参见图 2.16。while 循环主体的花括号缩进量应该大于程序 main 部分的花括号的缩进量。

至于在何处放置花括号，至少有两种理论。一种是像本书那样，每个花括号都单独占一行。这种形式更容易阅读。第二种是起始花括号不单独占一行。如果小心使用，第二种方法将更高效，而且更省空间。采用某种风格时，关键在于澄清程序结构。最终采用什么风格是因人而异的，你应该在自己的所有程序中坚持自己既定的风格。

注释

为使程序容易理解，应在程序的关键位置添加一些解释性批注。这些批注称为**注释**。C++和其他大多数编程语言一样都允许添加程序注释。在 C++中，符号//指出一条注释开始。符号//与行末之间的所有文本都是注释文本。编译器会忽略//之后这一行的所有内容。如果一条注释需要占据多行，就在每行注释之前添加符号//。符号//由两个连续的斜杠构成，之间无空格。

本书代码没有使用特殊格式，但在某些代码编辑器中，会使用和其他程序文本不同的颜色显示注释。

在 C++程序中，还有另一种方式插入注释。符号/*与*/之间的所有内容都会被视为注释，并被编译器忽略。/*到*/之间的注释可自由跨越多行。这样一来，就不需要像//注释那样，在每一行上都要添加一个额外的//。如下所示：

```
/* 这是一条跨越三行的注释。
注意第二行没有任何形式的
注释符号. */
```

① 格调、风格、样式在英语里面都是 Style。——译注

在程序中，但凡允许插入空格或换行符的位置，都允许插入/* */风格的注释。但不要把它插入不利于阅读的位置，也不要插入会干扰程序布局的位置。注释通常只应放在行末，或单独占一行。

至于哪种注释风格更好，人们的看法不一。但是，只要使用得当，任何一种注释都能取得理想的效果。本书统一采用//注释。

很难说程序中应该包含多少注释。唯一正确的答案是“够用就好”，但这对刚入门的程序员来说意义不大。只有积累一定经验之后，才能更好地把握添加注释的时机。一般在重要和不容易理解的地方添加注释。但注释太多有“喧宾夺主”之嫌。每行都有注释的程序使人难以看清程序的结构。例如，以下注释意义不大，不如不用：

```
distance = speed * time;  // 计算旅行的距离
```

注意图 2.17 的程序开头给出的注释。所有程序都应该以类似的注释开头。它给出了程序的所有基本信息：程序包含在哪个文件中、谁写的程序、如何联系作者、程序有何用途、程序的最新修改日期以及其他需要传达的信息，比如作业编号(如果该程序是课堂作业的话)。注释的具体内容要视情况而定。我们不打算在本书的其他程序中添加如此长的注释。但在你的程序中，应坚持在程序开头包含这样的注释。

图 2.17 注释和命名常量

```
 1  // 文件名：health.cpp(你的程序可能需要一个不同于 cpp 的扩展名)
 2  // 作者：<在此填写你的姓名>
 3  // 电子邮件：you@yourmachine.bla.bla
 4  // 题号：2
 5  // 说明：判断用户是否生病的程序
 6  // 上一次修改：2017 年 9 月 23 日
 7
 8  #include <iostream>
 9  using namespace std;
10  int main()
11  {
12      const double NORMAL = 98.6; // 正常体温(华氏度)
13      double temperature;
14
15      cout << "Enter your temperature: ";
16      cin >> temperature;
17
18      if (temperature > NORMAL)
19      {
20          cout << "You have a fever.\n";  // 你发烧了
21          cout << "Drink lots of liquids and get to bed.\n";  // 多喝水，卧床休息
22      }
23      else
24      {
25          cout << "You don't have a fever.\n";  // 你没有发烧
26          cout << "Go study.\n";  // 去上学
27      }
28
29      return 0;
30  }
```

程序坚持以这种注释开头

示范对话

```
Enter your temperature: 98.6
You don't have a fever.
Go study.
```

为常量命名

计算机程序的数字存在两个问题。第一是不好记。例如，数字 10 在程序中出现时，不能就其含义传达任何有意义的信息。例如在银行程序中，10 既可能是分行数量，也可能是总行出纳窗口数量。理解程序需知道每个常量的含义。第二是程序更改部分数字时容易出错。假定数字 10 在上述银行程序中出现了 12 次，其中 4 次代表分行数，另外 8 次代表总行出纳窗口数。一旦银行开设新的分行，并需要更新程序，有的 10 可能需要变成 11，有的根本不能改动。为了避免这些问题，一个办法是为每个数字命名，在程序中使用名称而不是数字本身。例如，可在银行程序中包含两个常量，分别命名为 BRANCH_COUNT(分行数)和 WINDOW_COUNT(窗口数)。两个常量的值可能都为 10，但在开设新分行时，只需更改 BRANCH_COUNT 的定义就可以了。

如何命名 C++程序中的数字呢？一个办法是将变量初始化为那个数字，示例如下：

```
int BRANCH_COUNT = 10;
int WINDOW_COUNT = 10;
```

但是以这种方式命名数字常量也有问题：你可能会不小心更改上述变量的值。C++提供了一种方式来标记初始化好的变量，使其不能更改。程序试图更改就会出错。要标记一个不能更改的变量，请在变量声明之前添加单词 const(constant 的缩写)。例如：

```
const int BRANCH_COUNT = 10;
const int WINDOW_COUNT = 10;
```

如果变量为同一类型，上述两行可合并为一个声明，如下所示：

```
const int BRANCH_COUNT = 10, WINDOW_COUNT = 10;
```

但大多数程序员都认为，每个名称定义单独占一行显得更清楚。const 这个词通常称为**修饰符**，因其修饰(限制)了被声明的变量。

使用 const 修饰符声明的变量称为**声明常量**或**命名常量**(named constant)。C++语言不要求声明常量的名称全部大写，但这已成了 C++程序员约定俗成的标准。

一旦像这样命名了数字，在需要使用该数字的任何地方，都可以使用它的名称。名称和数字本身具有完全相同的含义。更改命名常量只需更改 const 变量声明的初始值。例如，要将 BRANCH_COUNT 的值从 10 改为 11，只需修改 BRANCH_COUNT 声明的初始值 10。

尽管可以在程序中使用未命名的数字常量，但尽量少用。通常，只有已知的、容易识别的而且不会改变的量(比如“1 米等于 100 厘米”中的 100)才可以使用未命名的数字常量。但其他所有数字常量都应该像刚才描述的那样指定名称。这会使程序更容易阅读和修改。

图 2.17 用一个简单程序展示了如何使用 const 修饰符。

> **用修饰符 const 命名常量**
>
> 在声明中初始化变量时，可以标记变量，使程序不能更改该变量的值(使之成为常量)。为此，可在声明的开始处添加修饰符 const，如下所示：
>
> **语法**
>
> ```
> const Type Name Variable Name = Constant;
> ```
>
> **示例**
>
> ```
> const int MAX_TRIES = 3;
> const double PI = 3.14159265;
> ```

√ 自测题

35. 以下 if-else 语句能正确编译和运行。但风格和本章其他程序不一致。修改它使风格(缩进和换行)一致：

```
if (x < 0) {x = 7; cout << "x is now positive.";}
else {x = -7; cout << "x is now negative.";}
```

36. 以下两行语句的输出是什么(假定已把它们嵌入一个完整和正确的程序)?

```
// cout << "Hello from";
cout << "Self-Test Exercise";
```

37. 写完整的 C++程序，要求用户输入加仑数(gallon)，再输出等价的公升数(liter)。1 加仑等于 3.78533 公升。请使用一个声明常量。这只是练习题，不需要在程序中添加任何注释。

小　　结

- 为变量使用有意义的名称。
- 务必将变量声明为正确的数据类型。
- 变量使用前必须初始化。可在声明时初始化，也可在使用前用赋值语句初始化。
- 在算术表达式中合理使用圆括号，清楚指定运算顺序。
- 每次要求用户从键盘输入数据的时候都显示提示，并始终回显输入。
- if-else 语句允许程序在两个备选行动中选择一个。if 语句允许程序决定是否执行一个特定的行动。
- do-while 语句至少执行一次循环主体。而 while 循环有可能一次都不执行。
- 程序中，几乎所有数字常量都应被赋予有意义的名称，并在程序中使用名称代替数字。在变量声明中添加修饰符 const 来实现。
- 使用与本书示范程序类似的缩进、间距和换行风格。
- 插入注释来解释一个程序的重要小节或者任何不清楚的地方。

自测题答案

1.
```
int feet = 0, inches = 0;
    int feet(0), inches(0);
```

2.
```
int count = 0;
    double distance = 1.5;
```

   也可以使用：

```
    int count(0);
    double distance(1.5);
```

3. `sum = n1 + n2;`

4. `length = length + 8.3;`

5. `product = product * n;`

6. 这种程序的实际输出取决于系统和系统的使用历史。

```
#include <iostream>
using namespace std;
int main()
{
int first, second, third, fourth, fifth;
cout << first << " " << second << " " << third << " " << fourth << " " <<
fifth << endl;
    return 0;
}
```

7. 没有标准答案，以下答案仅供参考：
   a. speed
   b. payRate
   c. highest 或 maxScore

8.
```
cout << "The answer to the question of\n"
   << "Life, the Universe, and Everything is 42.\n";
```

9.
```
cout << "Enter a whole number and press return: ";
    cin >> theNumber;
```

10.
```
cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(3);
```

11.
```
#include <iostream>
using namespace std;
int main()
{
    cout << "Hello world\n";
    return 0;
}
```

12.
```
#include <iostream>
using namespace std;

int main()
    {
    int n1, n2, sum;
    cout << "Enter two whole numbers\n";
    cin >> n1 >> n2;
```

```
    sum = n1 + n2;
    cout << "The sum of " << n1 << " and "
         << n2 << " is " << sum <<endl;
    return 0;
}
```

13. `cout << endl << "\t";`

14.
```
#include <iostream>
using namespace std;
int main()
{
    double one(1.0), two(1.414), three(1.732), four(2.0), five(2.236);
    cout << "\tN\tSquareRoot\n";
    cout << "\t1\t" << one << endl
         << "\t2\t" << two << endl
         << "\t3\t" << three << endl
         << "\t4\t" << four << endl
         << "\t5\t" << five << endl;
    return 0;
}
```

15. `3*x`
`3*x + y`
`(x + y)/7`　　注意，`x + y/7` 是错误的。
`(3 * x + y)/(z + 2)`

16. bcbc

17. `(1/3) * 3 is equal to 0`

由于 1 和 3 是 int 类型，操作符/执行整数除法，会丢弃余数。因此，1/3 的值为 0，而不是 0.3333…。这使整个表达式变成 0 * 3，终值为 0。

18.
```
#include <iostream>
using namespace std;

int main()
{
    int number1, number2;
    cout << "Enter two whole numbers: ";
    cin >> number1 >> number2;
    cout << number1 << " divided by " << number2
         << " equals " << (number1/number2) << endl
         << "with a remainder of " << (number1%number2)
         << endl;
    return 0;
}
```

19. a. 52.0
b. 9/5 的值为 int 值 1。分子和分母都是 int，所以执行整数除法，小数部分会被丢弃。
c. `f = (9.0 / 5) * c + 32.0;` 或 `f = 1.8 * c + 32.0;`

20. 030406

这些字符串用操作符+连接到一起。

21.
```
if (score > 100)
    cout << "High";
    else
    cout << "Low";
```

可能要在以上双引号字符串的末尾添加\n，具体视程序的其他细节而定。

22.
```
if (savings >= expenses)
{
    savings = savings - expenses;
    expenses = 0;
    cout << "Solvent";
```

```
}
else
{
    cout << "Bankrupt";
}
```

可能要在以上双引号字符串的末尾添加\n，具体视程序的其他细节而定。

23.
```
if ( (exam >= 60) && (programsDone >= 10) )
      cout << "Passed";
    else
    cout << "Failed";
```

可能要在以上双引号字符串的末尾添加\n，具体视程序的其他细节而定。

24.
```
if( (temperature >= 100) || (pressure >= 200) )
      cout << "Warning";
    else
      cout << "OK";
```

可能要在以上双引号字符串的末尾添加\n，具体视程序的其他细节而定。

25. `(x <-1) || (x > 2)`

26. `(1 < x) && (x < 3)`

27. a. 输出 0 is false。在讲解类型兼容性的小节中，我们指出 int 值 0 会转换成 false。
b. 输出 1 is true。在讲解类型兼容性的小节中，我们指出非零的 int 值会转换成 true。
c. 输出-1 is true。在讲解类型兼容性的小节中，我们指出非零的 int 值会转换成 true。

28.
```
10
7
4
1
```

29. 没有任何输出，因为布尔表达式(x < 0)不满足，所以 while 语句直接终止，不执行循环主体。

30. 输出和自测题 28 一样。

31. 循环主体在检查布尔表达式之前执行。布尔表达式为 false，所以输出结果如下：

```
-42
```

32. 使用 do-while 语句，循环主体至少执行一次。使用 while 语句，循环主体可能一次都不执行。

33. 这是一个无限循环。输出将从以下内容开始，而且理论上会永远执行下去。

```
10
13
16
19
```

(一旦 x 的值超过计算机允许的最大整数，程序可能终止，或者产生一些奇怪的行为。但理论上这是一个无限循环。)

34.
```
#include <iostream>
using namespace std;

int main()
{
    int n = 1;
    while (n <= 20)
    {
        cout << n << endl;
        n++;
    }
    return 0;
}
```

```
35. if (x < 0)
{
    x = 7;
    cout << "x is now positive.";
}
else
{
    x = -7;
    cout << "x is now negative.";
}
```

36. 第一行是注释，编译器会将其忽略。所以整个输出如下所示：

```
Self-Test Exercise
```

37.

```
#include <iostream>
using namespace std;

int main()
{
    const double LITERS_PER_GALLON =3.78533;
    double gallons, liters;

    cout << "Enter the number of gallons:\n";
    cin >> gallons;

    liters = gallons* LITERS_PER_GALLON;
    cout << "There are " << liters << " liters in "
        << gallons << " gallons.\n";
    return 0;
}
```

编 程 练 习

编程练习一般只需写很小的程序，运用本章提到的编程概念。

1. 1 公吨(metric ton)等于 35273.92 盎司(ounce)。写一个程序，以盎司为单位读入一盒麦片(cereal)的重量，换算成公吨之后，输出这个重量，同时输出总共需要多少盒麦片才能凑足 1 公吨。你的程序应该允许用户任意重复计算。

2. 计算数字 *n* 的平方根的巴比伦算法如下所示：

   步骤 1：猜一个答案(第一次可以猜 *n*/2).，将值赋给 *guess*

   步骤 2：计算 $r = n / guess$

   步骤 3：$guess = (guess + r) / 2$

   步骤 4：回到步骤 2 重复。步骤 2 和 3 重复得越多，*guess* 就越接近 *n* 的平方根。

   写程序要求用户输入代表 *n* 的 `double` 值，使用巴比伦算法迭代 100 次。一个更有挑战性的版本是加入判断，如果当前 *guess* 和上一次 *guess* 相差不到 1%就输出答案。

3. 视频讲解：*Solution to Practice Program 2.3*

   许多跑步机在控制面板上显示英里/小时(mph)速度，但大多数跑步爱好者都用“步速”(pace)来衡量自己的跑步速度。步速一般是指跑一英里用了多少分钟多少秒，而不是使用 mph 作为单位。

   写程序以 mph 为单位输入一个值，将其换算为跑一英里的分秒数。例如，假定输入 6.5 mph，输出结果应该是跑一英里用了 9 分钟 13.8 秒。将 `double` 值转换为 `int`(丢弃小数点后的任何值)请使用：

```
intValue=static_cast<int>(dblVal);
```

4. 写程序玩 Mad Lib 游戏。提示用户输入以下字符串：

- 你的老师的姓或名(注意要么输入姓，要么输入名)
- 你的姓或名(注意要么输入姓，要么输入名)
- 一种食物
- 100 到 120 之间的一个数字
- 一个形容词
- 一种颜色
- 一种动物

输入这些字符串后，它们被代入下面的故事，并输出到控制台：

```
Dear Instructor,[老师的姓或名]

I am sorry that I am unable to turn in my homework at this time. First, I ate a rotten [食物], which made me turn [颜色] and extremely ill. I came down with a fever of [100～120]. Next, my [形容词] pet [动物] must have smelled the remains of the [食物] on my homework, because he ate it. I am currently rewriting my homework and hope you will accept it late.

Sincerely,
[你的姓或名]
```

如果愿意写一个中文版的程序，请使用以下故事模板[①]：

亲爱的[老师姓名]老师：

　　对不起，我暂时交不了作业。我先是吃了一个烂掉的[食物]，我全身都变[颜色]了，感觉很糟糕，现在发烧[100~120]度。然后，我的[形容词]宠物[动物]肯定是闻到了作业本上的[食物]的味道，因为它居然把作业本给吃掉了。现在我正在重新写作业，希望晚些时候能交上来。

你的忠实的[你的姓名]

5. 以下程序计算给定半径的球体的体积。能编译和运行，但不符合 2.5 节推荐的编程风格。请根据本章描述的风格重写程序，添加缩进和注释，并使用合理命名的常量。

```
#include <iostream>
using namespace std;
int main() {
    double radius, vm;
    cout << "输入球体半径: " << endl; cin >> radius;
    vm = (4.0 / 3.0) * 3.1415 * radius * radius * radius;
    cout << " 体积是: " << vm << endl;
    return 0;
 }
```

编 程 项 目

编程项目要求综合运用多方面的知识来解决问题，程序一般比编程练习大，解题方式多样化。

1. 一家政府研究机构的研究表明，减肥碳酸饮料中常用的一种人造甜味剂会导致实验室里的小白鼠死亡。你的一位朋友很想减肥，不肯放弃可能导致死亡的碳酸饮料。她想知道，在不危及生命的情况下，可以喝多少碳酸饮料。写一个程序来提供答案。需要向程序输入让一只小白鼠致死所需的人造甜味剂重量(5 克)、小白鼠体重(35 克)和节食者预期的体重(如输入磅数，记住 1 磅等于 454 克)。假定致鼠死亡的剂量和致人死亡的剂量成正比。一罐碳酸饮料的重量假定是 350 克。为保证朋友的安全，程序一定要提示输入停止减肥时的体重，而不要输入当前体重。假定碳酸饮料含 1‰(千分之一)人造甜味剂，在表示这个比例的变量声明中使用修饰符 const。注意，可能需要将这个比例表示成 double 值 0.001。程序应该允许用户任意重复计算。

① 中文姓名可以全部输入，比如可以输入“张三”，而不必像英文版那样，只能输入 Zhang 或者 San。但无论如何，姓和名之间不能有空格。——译注

2. 某公司决定为员工普涨 7.6%的工资，同时按新标准补发前 6 个月工资。写程序输入员工去年年薪，输出应补发金额、新的年薪和新的月薪。在表示涨幅的变量声明中使用修饰符 const。程序应该允许用户任意重复计算。

3. 修改编程项目 2 的程序，让它能计算补发任意月数的工资，而不是固定补发 6 个月。具体月数由用户输入。

4. 消费贷款的计算并不一定简单。有一种贷款方式称为“贴现式分期贷款”(discount instalment loan)，它的计算方式如下：假设贷款金额为 1000 美元，年利率为 15%，贷款期限为 18 个月。贷款金额 1000 美元乘以 0.15，年利息是 150 美元。它乘以贷款期限 1.5 年(18 个月)，总利息是 225 美元。这个金额立即从 1000 美元中扣除，消费者实际拿到手上的贷款只有 775 美元。还贷时，消费者还是以贷款金额为准，每月偿还相同的金额。所以，月还贷额是 1000 美元除以 18，即 55.56 美元。如果消费者需要的恰好是 775 美元，那么很快就能完成计算。但是，如果消费者需要的是 1000 美元，人工计算就显得比较繁琐。请写一个程序来获取三项输入：消费者想实际拿到手的金额、利率和贷款期限(以月为单位)。程序要计算需要多大的贷款金额，消费者才能拿到所需的金额。程序还应该计算月还款额。程序应该允许用户任意重复计算。

5. 写程序判断一间会议室是否违反了消防条例中有关房间最大容积(最多容纳人数)的条款。程序将读取房间最大容积和与会人数。如果人数小于或等于最大容积，程序宣布会议合法，并指出还能容纳多少人。人数超过最大容积，程序宣布依据消防条例，会议不能举行，并指出要减去多少人才能符合消防条例的要求。可为这个程序写一个更复杂的版本，允许用户任意重复计算。如果这是课堂作业，请询问教师是否需要写复杂版本。

6. 某公司规定，在每周的正常上班时间中(40 小时)，员工工资标准是每小时 16.78 美元。超过正常时间算加班。每加班 1 小时，工资是正常标准的 1.5 倍。在员工的应发工资中，缴纳 6%作为社会保险税，14%作为美国联邦税，5%作为美国州税，再加上每周 10 美元的工会会费。假如员工有 3 名或更多家属，还要多缴 35 美元的额外医疗保险费。写程序读取员工在某一周的工作时数及其家属人数，输出员工在那一周的应发工资、每一笔代扣金额和最后拿到手的实发工资。可为此程序写一个更复杂的版本，允许用户任意重复计算。如果这是课堂作业，请询问教师是否需要写复杂版本。

7. 制定跨度为几年的预算并不容易，因为价格并非一成不变。如果你的公司每年需要 200 支铅笔，就不能简单地根据今年的价格来计算今后两年的铅笔开支。由于通货膨胀，开支可能比现在高。写程序来估算在指定的年数之后，某件商品的预计开支。程序要求输入今年购买每一件商品的开支、从现在起的年数和通货膨胀率。然后，程序输出在经过那么多年之后，每一年购买同一件商品所需的开支。要求用户以百分数的形式输入通货膨胀率，比如 5.6(它代表 5.6%)。随后，程序将百分数转换成小数，如 0.056，并用一个循环来估算根据通货膨胀率而得出的开支(**提示**：这类似于计算信用卡账户的利息，详情参见本章前面的讨论)。

8. 假设分期贷款购买一套立体声音响，价格 1000 美元，还贷计划如下：无首付款，年利率 18%(月利率 1.5%)，每月还款 50 美元。在这 50 美元中，扣除利息之后，剩下的金额用于偿还本金。所以，第一个月支付的利息是 1000 美元的 1.5%，即 15 美元，偿还的本金则为 35 美元。现在，贷款额从 1000 美元变成 965.00 美元。第二个月支付的利息为 965.00 美元的 1.5%，即 14.48 美元。50 美元减去 14.48 美元得到 35.52 美元，这是第二个月偿还的本金。写一个程序，计算还清贷款的期限和在这期间支付的总利息额。使用一个循环来计算利息和每月还款之后的剩余贷款(最终的程序不需要输出每月应支付多少利息和剩余贷款，但你可考虑写程序的一个增强版本，从而输出这些值)。使用一个变量对循环迭代进行计数，它的终值就是还清贷款(剩余贷款减至 0)所需的月数。可能还需要使用其他变量。如果剩余贷款已经不多，那么最后一笔还款可能小于 50 美元(但不要忘记还有利息)。如果尚余 50 美元的贷款额，那么月付 50 美元还不能还清贷款，虽然此时已非常接近还清。注意，对于 50 美元的贷款，月利息仅为 75 美分。

9. 写程序读入 10 个整数，输出大于 0 的所有整数(正数)之和、小于 0 的所有整数(负数和 0)之和与所有整数(无论正数、负数还是 0)之和。用户可一次性输入这 10 个整数，而且可以采用任何顺序。程序不应该要求用户单独输入正数和负数。

10. 修改编程项目 9 的程序，输出所有正整数之和、所有正整数的平均数、所有负整数之和、所有负整数的平均数、所有正整数和负整数之和以及所有数的平均数。

11. 声音在空气中通过分子之间的碰撞来传输。气温影响分子速度，进而影响音速。干燥空气中的音速大致可以使用以下公式：

$$v \approx 331.3 + 0.61 \times T_c$$

其中的 T_c 是以摄氏度为单位的空气温度，而 v 是以 m/s 为单位的音速。

写程序允许用户输入一个起始和结束温度。在这个温度范围内，程序应输出每一档温度和对应的音速，每一档都提高 1℃。例如，假定起始和结束温度分别是 0℃和 2℃，那么程序输出如下：

英文版：
At 0 degrees Celsius the velocity of sound is 331.3 m/s
At 1 degrees Celsius the velocity of sound is 331.9 m/s
At 2 degrees Celsius the velocity of sound is 332.5 m/s

中文版：
0℃时，音速为 331.3 m/s
1℃时，音速为 331.9 m/s
2℃时，音速为 332.5 m/s

12. 视频讲解：*Solution to Programming Project 2.12*

许多私人打的水井每分钟只能出 1 或 2 加仑的水。为了防止没水的情况，一般的办法是安装临时储水舱。一家四口每日用水 250 加仑。但是，井本身就是一个“自然”的储水舱。挖得越深，储水越多，甚至会多过家庭的用水量。但是，具体有多少水可供使用？

写程序让用户以英寸为单位输入水井半径(典型水井半径为 3 英寸)，以英尺为单位输入水井深度(假设水充满整个深度，虽然这实际上不可能，因为水平面一般低于地平面 50 英尺以上)。程序输出水井的储水量(加仑)。为方便你参考，圆柱体体积公式是$\pi r^2 h$，其中 r 是半径，h 是高度。另外，1 立方英尺=7.48 加仑，1 英尺=12 英寸。

例如，半径 3 英寸、深 300 英尺的水井在充满水的情况下，可容纳 441 加仑的水，足够一家四口使用，不需要安装单独的储水缸

13. Harris–Benedict 公式估算在不锻炼的情况下保持体重所需的热量(卡路里)。这称为基础代谢速率(basal metabolic rate，BMR)。

女性保持体重所需的热量使用以下公式：

BMR = 655 + (4.3 × 体重磅数) + (4.7 × 身高英寸数) – (4.7 × 年龄)

男性保持体重所需的热量使用以下公式：

BMR = 66 + (6.3 × 体重磅数) + (12.9 × 身高英寸数) – (6.8 × 年龄)

普通巧克力条提供 230 卡路里热量。写程序让用户输入体重(以磅为单位，100 磅等于 45.4 公斤)、身高(以英寸为单位，1 英寸等于 2.54 厘米)和年龄。输入字符 M 代表男性，输入 F 代表女性。输出每天要吃多少巧克力条才能保持体重。

14. 写程序计算 N 次课堂作业的总成绩，然后用百分数输出该成绩。用户首先输入 N 的值，表明总共有多少次课堂作业。接着输入每次作业的实际成绩(Score received for exercise)和满分成绩(Total points possible for exercise)。所有作业的实际成绩除以满分成绩，获得总成绩。以百分数形式输出。下面是示例输出：

```
How many exercises to input? 3

Score received for exercise 1: 10
Total points possible for exercise 1: 10
```

```
Score received for exercise 2: 7
Total points possible for exercise 2: 12

Score received for exercise 3: 5
Total points possible for exercise 3: 8

Your total is 22 out of 30, or 73.33%.
```

15. 为适应气候变化，建筑必须考虑热膨胀效应。例如，金属顶梁高温膨胀。附加应力可能造成建筑受损。类似地，材料低温收缩。对于一种能自由膨胀的材料，其长度的线性变化公式如下所示：

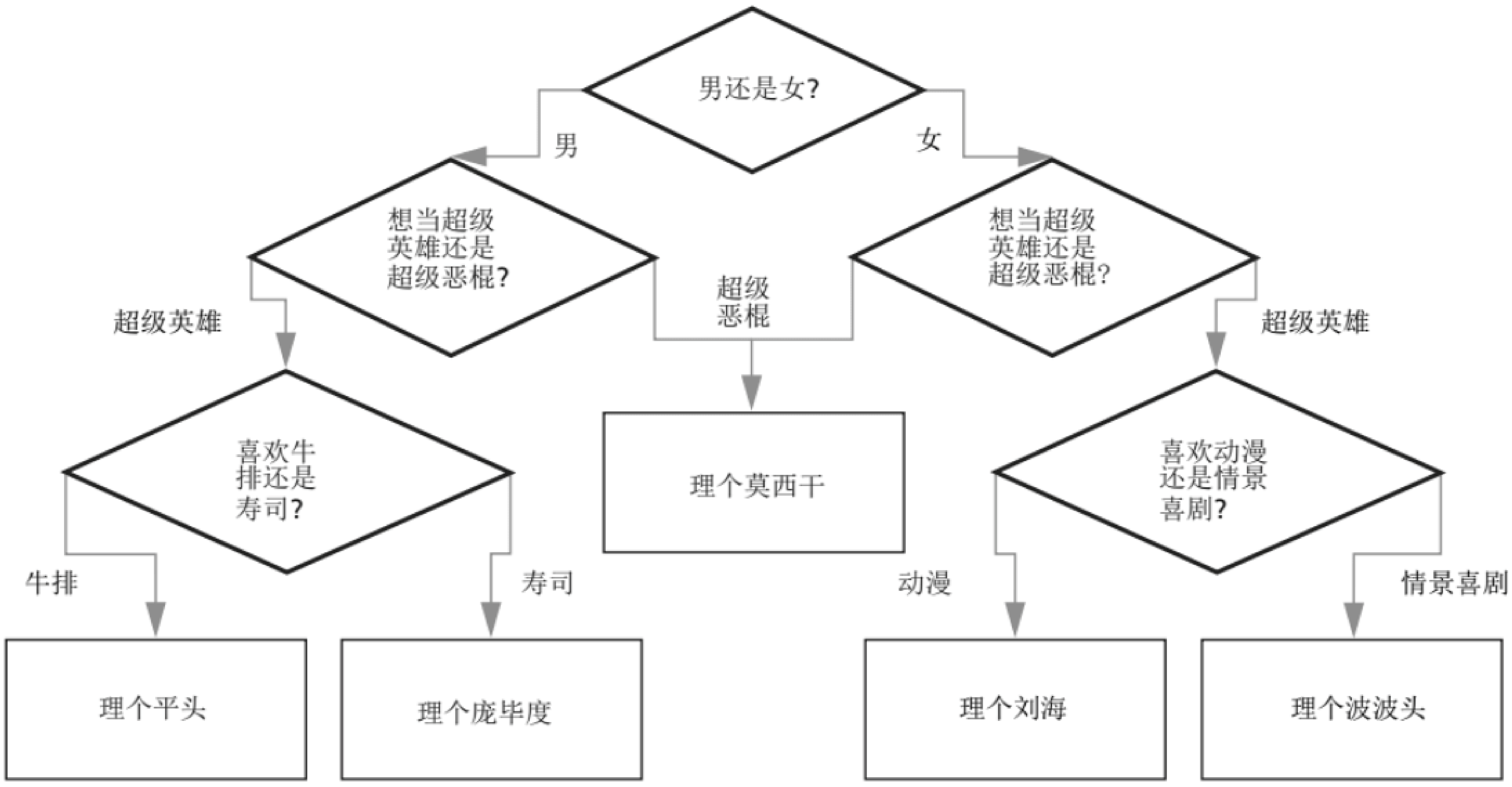
$$L_{\Delta} = \alpha L_0 T_{\Delta}$$

其中，L_0是材料初始长度(米)，L_{Δ}是线性位移(米)，T_{Δ}是温度变化(℃)，而α是线性热膨胀系数。

写程序输入α，L_{Δ}和 T_{Δ}，计算并输出线性位移。如位移为正，就输出“The material will expand by *x* meters”(将膨胀 *x* 米)。为负就输出“The material will contract by *x* meters”(将收缩 *x* 米)，此时的 *x* 应显示成负数。下面列出了一些常见材料的α值：

铝	2.31×10^{-5}
铜	1.70×10^{-5}
玻璃	8.50×10^{-6}
钢	1.20×10^{-5}

16. 下图通过一系列问题决定发型。写程序问问题并输出推荐发型①。



① 译者特意为好奇的读者提供以下参考。另外，典型情景喜剧包括《生活大爆炸》和《破产姐妹》等。

莫西干



头顶两边光秃秃，中间头发向上翘起好似马鬃

平头



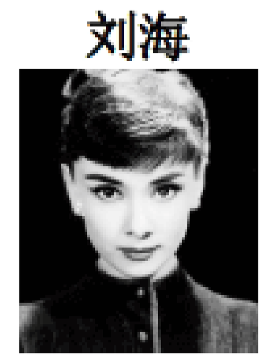
从脑后到两鬓的头发全部推光，上端头发稍长齐平

庞毕度



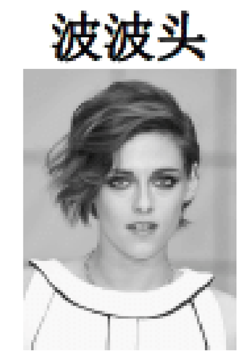
刘海向后梳，前面头发微微膨起

刘海



从前额垂下一些头发，遮住额头或脸部的一部分

波波头



齐耳短发

第 3 章　更多的控制流程

当您来到人生的岔路口时，请好好把握。

——*尤吉·贝拉*

概述

程序语句执行顺序称为**控制流程**或**控制流**。if-else，while 和 do-while 语句是指定控制流程的三种方式。本章将探索使用这些语句的一些新方式，并另外介绍两个新语句：switch 和 for 语句。if-else，while 和 do-while 语句的行动由布尔表达式控制。首先更深入地讨论布尔表达式。

预备知识

本章基于第 2 章的知识。

3.1　使用布尔表达式

“正好相反，”叮当弟接着说，“如果那是真的，那就可能是真的；如果那曾经是真的，它就是真的过；但既然现在它不是真的，就说明现在它是假的。这是逻辑。”

——*刘易斯·卡洛尔，《爱丽丝镜中奇遇记》*

布尔表达式求值

布尔表达式是求值为真(true)或假(false)的表达式，true 表示条件满足，false 表示条件不满足。以前曾在 if-else 语句中将布尔表达式用作测试条件，在 while 等循环语句中用作控制表达式。但布尔表达式本身是独立的，不依附于 if-else 或循环语句。利用 C++类型 bool，可声明值为 true 或 false 的变量。

布尔表达式可采取与算术表达式一样的方式求值。两者唯一的区别在于，算术表达式执行+、-、*和 / 等运算，结果是数字。相反，布尔表达式执行==和<等关系运算，以及&&、|| 和 ! 等布尔运算，结果是 true 或 false。==、!=、<和<=等运算作用于任何内建类型的一对值，并生成布尔值 true 或 false。理解了布尔表达式的求值方式后，就能写出并理解复杂的布尔表达式，并可为函数返回值使用布尔表达式。

首先分析算术表达式的求值。同样的技术也适用于布尔表达式的求值。例如以下算术表达式：

```
(x + 1) * (x + 3)
```

假定变量 x 的值为 2。为了对这个算术表达式进行求值，需要进行两次求和运算，获得数字 3 和 5，然后用操作符*求两个数的乘积，终值为 15。注意在求值过程中，并不是让表达式 (x + 1) 和 (x + 3) 相乘，而是让两个表达式的值相乘。执行乘法运算时，实际使用的是 3 和 5，而不是 (x + 1) 和 (x + 3)。

计算机采取相同的方式对布尔表达式进行求值。子表达式先求值，结果为 true 或

false。这些独立的 true 或 false 值根据图 3.1 的规则进行合并。例如以下布尔表达式：

```
!( (y < 3) || (y > 7) )
```

它也许是一个 if-else 语句或者 while 语句的控制表达式。假定 y 值为 8，那么在本例中，(y< 3)求值为 false，(y > 7)求值为 true，所以上述布尔表达式等价于：

```
!( false || true)
```

参考图 3.1(|| 在其中标记为 OR)，计算机知道圆括号内的表达式应求值为 true。所以，计算机知道整个表达式等价于以下表达式：

```
!(true)
```

再次参考图 3.1，计算机知道!(true)求值为 false，所以它知道 false 是原始布尔表达式的终值。

图 3.1　真值表

AND

Exp_1	Exp_2	Exp_1 && Exp_2
true	true	true
true	false	false
false	true	false
false	false	false

OR

Exp_1	Exp_2	Exp_1 || Exp_2
true	true	true
true	false	true
false	true	true
false	false	false

NOT

Exp	!(Exp)
true	false
false	true

以前几乎所有例子都完整地添加了圆括号，准确指出每个&&、|| 和! 应用于哪一个表达式。但圆括号并非总是必要的。如省略圆括号，默认优先级就是：首先执行!，然后对<之类的关系操作符进行求值，然后对&&进行求值，再对 || 进行求值。但作为良好编程习惯，最好保留大多数圆括号，使表达式更容易理解。在一连串&&或||(但两者不能混用)中，可以放心地省略圆括号。以下表达式不仅是有效的，还具有良好的可读性：

```
(temperature > 90) && (humidity > 0.90) && (poolGate == OPEN)
```

由于关系运算>和==先于&&进行求值，所以可省略上述表达式中的圆括号。两种做法的效果虽然相同，但如果包括一些圆括号，将有助于改善代码的可读性。

圆括号从表达式中删除以后，计算机会根据**优先级规则**对各个项进行分组。图 3.2 总结了 C++的部分优先级规则。如果一个运算先于另一个运算进行求值，就认为首先求值的运算具有**较高的优先级**。优先级相同的二元运算从左向右求值，优先级相同的一元运算从右向左求值。完整优先级规则请参见附录 2。

图 3.2　优先级规则

一元操作符+，-，++，--和!　　　　**最高的优先级(最先求值)**
二元算术操作符*，/，%
二元算术操作符+，-
布尔运算<，>，<=，>=
布尔运算==，!=
布尔运算&&
布尔运算||　　　　**最低的优先级(最后求值)**

注意，优先级规则中同时包括算术操作符(比如+和*)和布尔操作符(比如&和||)。这是由于许多表达式合并了算术运算和布尔运算，如下例所示：

```
(x + 1) > 2 || ( x + 1) < -3
```

参考图 3.2 的优先级规则，就知道此表达式等价于以下表达式：

```
((x + 1) > 2) || ((x + 1) < -3)
```

这是因为>和<的优先级高于 ||。事实上，完全可以省略上述表达式中的所有圆括号，含义是一样的，只是阅读起来比较困难。虽然不提倡省略所有圆括号，但可以通过这种方式理解表达式如何用优先级规则来解释。下面是删除了所有圆括号之后的表达式：

```
x + 1 > 2 || x + 1 < -3
```

根据优先级规则，首先应用一元操作符-，然后应用+，然后应用>和<，最后应用 ||，含义和保留完整圆括号的版本一样。

前面对布尔表达式求值方式的描述基本正确，但在 C++中，计算机在对布尔表达式进行求值时，偶尔会走一条捷径。许多时候只需求值布尔表达式的第一个子表达式。例如：

```
(x >= 0) && (y > 1)
```

如果 x 为负数，(x >= 0)就为 false，按照图 3.1 所总结的，&&表达式的一个子表达式为 false，整个表达式都为 false，不管另一个表达式为 true 还是 false。所以，如果知道第一个表达式为 false，就没必要求值第二个表达式。|| 表达式也会发生类似的情况。在用操作符||连接的两个表达式中，第一个表达式为 true，整个表达式肯定为 true，不管第二个表达式为 true 还是 false。C++语句会利用这一事实，在由&&和 || 连接的逻辑表达式中避免对第二个子表达式进行多余的求值。C++首先求值最左边的子表达式。如果获得的信息足以判断整个表达式的终值(不管第二个子表达式的值是什么)，C++就不再对第二个子表达式求值。这种求值方法称为**短路求值**。

和 C++不同，一些语言采用**完全求值**。在完全求值中，如果两个子表达式用&&或||连接，两个子表达式都会求值，最后根据真值表确定整个表达式的终值。

无论短路求值还是完全求值，答案都是相同的，那么为什么还要关心 C++所用的短路求值呢？事实上，大多数时候都不必关心这一点。只要由&&或 || 连接的两个子表达式都有一个值，两种方法会产生相同的结果。然而，假如第二个子表达式是未定义的，就能立即体现出 C++的短路求值的好处。下面对此进行了演示，首先分析这个语句：

```
if ( (kids != 0) && ((pieces/kids) >= 2) )
    cout << "Each child may have two pieces!";
```

如果 kids 的值不为 0，上述语句没什么好说的。但是，如果 kids 的值为 0，考虑一下短路求值能带来什么好处。表达式(kids != 0)首先求值为 false，所以没有必要再对第二个表达式进行求值。使用短路求值，C++认为整个表达式已经为 false，不必多此一举求值第二个表达式。这就防止了一个运行时错误，因为第二个表达式在求值时会产生除以 0 错误。

C++有时将整数当作布尔值使用。具体说来，C++会将整数 1 转换成 true，将整数 0 转换成 false。另外，实际情况并非只是“将 1 用作 true，将 0 用作 false”那么简单。编译器会将所有非 0 的数视为 true，将 0 视为 false。只要正确编写布尔表达式，这种自动转换不会造成任何问题，甚至根本不需要注意到这个转换。但在调试程序时，一定要记住编译器允许使用布尔操作符&&、|| 和 ! 来连接整数。

> **布尔(bool)值就是 true 和 false**
>
> 在 C++中，条件成立，布尔表达式求值为 bool 值 true；条件不成立，布尔表达式求值为 bool 值 false。

陷阱：将布尔表达式转换成 int 值

假定要在 if-else 语句中使用布尔表达式，时间未用完之前为 true(比如某些游戏或者处理过程)。更准确地说，要在一个 if-else 语句中使用布尔表达式，只要 int 类型的变量 time 不大于变量 limit 的值，该表达式就为 true。为此，你可能像下面这样写程序(*Something* 和 *Something_Else* 代表一些 C++语句)：

```
if (!time > limit)    ←—— 这与本意不符
    Something
else
    Something_Else
```

把它大声地念出来，听起来似乎没有问题：“time 不大于 limit”。但布尔表达式的写法错了；更糟的是，编译器不显示错误消息。这样写表明还没有吃透 C++的优先级规则。对于上述语句，编译器会应用图 3.2 的优先级规则，对其中的布尔表达式进行如下解释：

```
(!time) > limit
```

这看起来就像在说胡话，完全有悖于直觉。例如，假定 time 的值是 36，那么(!time)是什么意思呢？“非 36”？。在 C++中，任何非 0 的整数都转换成 true，0 转换成 false。所以，!36 被解释成“not true”(非真)，也就是 false。由于要和一个 int 值进行比较，所以 false 会被转换回 0。

所以，这个布尔表达式的实际结果和我们的本意是不符的。如 time 值为 36，而 limit 值为 60，本意是让布尔表达式求值为 true(因 time > limit 不成立)。遗憾的是，这个布尔表达式实际这样求值：(!time)求值为 false，转换成 0，所以整个布尔表达式等价于：

```
0 > limit
```

也就是 0 > 60，最终求值结果为 false(本意是求值为 true)。

有两个办法解决这个问题。一个是正确使用操作符!。使用!时，一定要用圆括号来封闭需要求反的参数。上述布尔表达式的正确写法如下：

```
if (!(time > limit))
    Something
else
    Something_Else
```

另一个办法是彻底避免使用操作符!。例如，以下语句同样正确，而且更明白：

```
if (time <= limit)
    Something
else
    Something_Else
```

操作符!几乎总是可以避免的，这同时是许多程序员提倡的。他们认为，英语中的 not 会使一件事情变得不容易明白。同样，“not”操作符! 会使 C++程序变得不容易明白。当然，也没必要刻意避免使用操作符 !，但在使用它之前，确实应该想一想，是否能在不使用它的前提下，更清楚地表达同一件事情。 ■

bool 类型是新增的

老版本 C++不支持 bool 类型，而是用整数 1 和 0 表示 true 和 false。如果你用的正是老版本的、不支持 bool 类型的 C++，请立即换用新编译器。

√ 自测题

1. 判断以下每个布尔表达式的值为 true 还是 false，假定变量 count 的值为 0，变量 limit 的值为 10。答案必须是 true 或 false 这两个值之一。

   a. `(count == 0) && (limit < 20)`
   b. `count == 0 && limit < 20`
   c. `(limit > 20) || (count < 5)`
   d. `!(count == 12)`
   e. `(count == 1) && (x < y)`
   f. `(count < 10) || (x < y)`
   g. `!( ((count < 10) || (x < y)) && (count >= 0) )`
   h. `((limit/count) > 7) || (limit < 20)`
   i. `(limit < 20) || ((limit/count) > 7)`
   j. `((limit/count) > 7) && (limit < 0)`
   k. `(limit < 0) && ((limit /count) > 7)`
   l. `(5 && 7) + (!6)`

2. 指出 C++中能改变执行顺序的两种语句。请给出一些例子。

3. 在高等代数中，我们知道数值区间可如下表示：

   ```
   2 < x < 3
   ```

   在 C++中，同样的表示法却不能定义你所希望的区间。请解释为什么，并给出正确的 C++布尔表达式以指定 *x* 的取值范围为 2～3。

4. 以下语句会产生除以 0 错误吗?

   ```
   j = -1;
   if ((j > 0) && (1/(j+1) > 10))
       cout << i << endl;
   ```

枚举类型(选读)

枚举类型是值用一组 int 类型的常量来定义的类型。枚举类型就像是包含了一组声明常量的列表。

定义枚举类型时可以使用任何 int 值，并可在枚举类型中定义任何数量的常量。例如，以下枚举类型为每个月的长度(天数)定义了一个常量：

```
enum MonthLength { JAN_LENGTH = 31, FEB_LENGTH = 28,
MAR_LENGTH = 31, APR_LENGTH = 30, MAY_LENGTH = 31,
JUN_LENGTH = 30, JUL_LENGTH = 31, AUG_LENGTH = 31,
SEP_LENGTH = 30, OCT_LENGTH = 31, NOV_LENGTH = 30,
DEC_LENGTH = 31 };
```

从中看出，在枚举类型中，两个或更多的命名常量可接收同一个 int 值。

如果不指定任何数值，枚举类型中的标识符会被自动分配一组连续的值，这些值从 0 开始。例如以下类型定义：

```
enum Direction { NORTH = 0, SOUTH = 1, EAST = 2, WEST = 3 };
```

等价的定义如下：

```
enum Direction { NORTH , SOUTH, EAST, WEST };
```

如果希望的只是一组名称，不关心它们的值，就建议不显式地列出 int 值。

如果只将部分枚举常量初始化成特定的值，例如以下定义：

```
enum MyEnum { ONE = 17, TWO, THREE, FOUR = -3, FIVE };
```

那么 ONE 将获得值 17，TWO 获得下一个 int 值 18，THREE 获得下一个 int 值 19，FOUR 获得-3，FIVE 获得下一个值-2。

简单地说，第一个枚举常量的默认值是 0，其余常量依次递增 1——除非显式指定了一个或者多个枚举常量的值。

C++11 引入了一种新的枚举，称为**强枚举**或**枚举类**。它避免了传统枚举存在的一些问题。例如，你可能不希望枚举具有整数的行为。另外，传统枚举具有全局作用域，所以不允许有两个相同的枚举值。为了定义强枚举，在 enum 后添加关键字 class 就可以了。枚举名称后添加两个冒号来限定要使用的枚举值。例如：

```
enum class Days { Sun, Mon, Tue, Wed };
enum class Weather { Rain, Sun };
Days d = Days::Tue;
Weather w = Weather::Sun;
```

3.2 多 路 分 支

“请你告诉我，离开这里应该走哪条路？”

“这要看你想上哪儿去。”猫说。

——*刘易斯·卡洛尔，《爱丽丝漫游奇境记》*

任何编程构造，只要能从多个备选行动中选择一个，就是一种**分支机制**。if-else 语句能从两个备选行动中选择一个。本节讨论能从两个以上备选行动中做出选择的分支机制。

嵌套语句

如前所述，if-else 和 if 语句中都包含较小的语句。这些较小的语句可以是复合语句和简单语句(如赋值语句)，但还有其他选择。事实上，任何语句都能作为 if-else、if、while 或者 do-while 语句的一个子部分来使用。图 3.3 对此进行了演示。从方框的形态可以看出，语句进行了三级嵌套。两个 cout 语句嵌套在一个 if-else 语句中，而这个 if-else 语句又嵌套在一个 if 语句中。

每一级嵌套通常都应该缩进。图 3.3 进行了三级嵌套，也进行了三级缩进。两个 cout 语句的缩进量相同，因为它们在同一个嵌套级别上。本章后面会介绍其他缩进模式。但是，除非遇到特殊情况，否则都应该像图 3.3 那样缩进每个嵌套级别。

图 3.3　嵌套在 if 语句中的 if-else 语句

```
1  if (count > 0)
2      if (score > 5)
3          cout << "count > 0 and score > 5\n";
4      else
5          cout << "count > 0 and score <= 5\n";
```

编程提示：在嵌套语句中使用花括号

假定要写一个 if-else 语句，将其用于赛车的板载计算机监视系统。程序的这个部分能在油量偏低时警告赛车手，但在油箱接近满载时告诉赛车手绕过维修站①。在其他所有情况下，程序都不提供任何输出，以免赛车手分心。我们设计了以下伪代码：

如果油量计(fuel gauge)不到满箱的 3/4，那么：
　检查油量计是否低于满箱的 1/4；如果是，就发出低油量警告。
否则(也就是说，油量计高于满箱的 3/4)：
　输出一条消息，告诉赛车手不要停车。

一个粗心的人可能像下面这样实现伪代码：

```
if (fuelGaugeReading < 0.75)
    if (fuelGaugeReading < 0.25)
        cout << "Fuel very low, Caution!\n";
else
    cout << "Fuel over 3/4, Don't stop now!\n";
```

阅读正文，了解错在哪里

上述实现看起来没错，编译器也认为正确，所以不会报错。但是，它并没有真正实现上述

① 即 pit stop，通常要由 21 个人协作，花 6～12 秒的时间为赛车加气、加油和换胎。——译注

伪代码。注意，if 在其中出现了两次，但 else 只出现一次。编译器必须判断哪个 if 与 else 配对。上述代码进行了良好的缩进，显示出 else 应该与第一个 if 配对，但编译器并不关心你如何缩进。对于编译器来说，上述嵌套语句等价于以下版本，它和上个版本唯一的区别就是缩进的方式：

```
if (fuelGaugeReading < 0.75)
    if (fuelGaugeReading < 0.25)
        cout << "Fuel very low, Caution!\n";
    else
        cout <<"Fuel over 3/4, Don't stop now!\n";
```

遗憾的是，编译器会采用第二种解释，使 else 与第二个 if(而不是第一个 if)配对。这称为 **else 虚悬问题**，图 3.4 的程序对此进行了说明。

编译器总是将 else 和上一个未与其他 else 配对的 if 进行配对。但是，不要把希望全部寄托在规则上。忘记规则！改变规则！你才是老大！总是告诉编译器你想如何做，确保编译器按你的意愿行事。那么，如何将自己的意图传达给编译器呢？请使用花括号。嵌套语句中的花括号就像算术表达式中的圆括号。花括号告诉编译器如何对不同项目进行分组，而不是按照默认约定进行分组，后者也许达不到你所希望的效果。为了避免问题，并使你的程序更易读，请围绕 if-else 语句中的子语句添加一对花括号，就像图 3.4 为第一个 if-else 语句所做的那样。

对于非常简单的子语句，比如单个赋值语句或者单个 cout 语句，则可以放心地忽略花括号。在图 3.4 中，以下子语句(在第一个 if-else 语句中)的花括号可以省略：

```
cout << "Fuel over 3/4, Don't stop now!\n";
```

然而，即使在这些简单情况下，花括号有时也能增强可读性。有的程序员建议为 if-else 语句中即使最简单的子语句添加花括号，就像图 3.4 为第一个 if-else 语句所做的那样。

图 3.4　花括号的重要性

```
1  // 演示花括号在 if-else 语句中的重要作用
2  #include <iostream>
3  using namespace std;
4  int main()
5  {
6     double fuelGaugeReading;
7
8     cout << "Enter fuel gauge reading: ";
9     cin >> fuelGaugeReading;
10
11    cout << "First with braces:\n";
12    if (fuelGaugeReading < 0.75)
13    {
14       if (fuelGaugeReading < 0.25)
15          cout << "Fuel very low. Caution!\n";
16    }
17    else
18    {
19       cout << "Fuel over 3/4. Don't stop now!\n";
20    }
21
22    cout << "Now without braces:\n";
23    if (fuelGaugeReading < 0.75)
24       if (fuelGaugeReading < 0.25)        <-- 这种缩进方式很好，但
25          cout << "Fuel very low. Caution!\n";     计算机不会注意到这些
26    else
27       cout << "Fuel over 3/4. Don't stop now!\n";
28
```

```
29      return 0;
30  }
```

示范对话 1

```
Enter fuel gauge reading: 0.1
First with braces:
Fuel very low. Caution!
Now without braces:
Fuel very low. Caution!
```

花括号在这种情况下没有造成什么差异，但请看示范对话 2

示范对话 2

```
Enter fuel gauge reading: 0.5
First with braces:
Now without braces:
Fuel over 3/4. Don't stop now!
```

这里不应该有任何输出，多亏有了花括号

没有花括号，这个版本产生了错误的输出

多路 if-else 语句

if-else 语句属于两路分支，允许程序从两个备选行动中选择一个。但是，经常都需要 3 路或 4 路分支，使程序能在两个以上的备选行动中选择一个。可通过嵌套 if-else 语句实现这种多路分支。例如，假定要设计一个游戏程序，用户必须猜中一个数字。数字保存在名为 number 的变量中，用户猜测的数字保存在变量 guess 中。如果希望在每次猜测之后都给用户一个提示，可以设计以下伪代码：

```
在 guess > number 的时候输出 "Too high."
在 guess < number 的时候输出 "Too low."
在 guess == number 的时候输出 "Correct!"
```

任何时候，只要分支操作被描述成一组相互排斥的条件及其对应的行动(就像本例一样)，就可使用嵌套 if-else 语句来实现该操作。例如，上述伪代码可转换成以下 C++代码：

```
if (guess > number)
    cout << "Too high.";
else if (guess < number)
    cout << "Too low.";
else if (guess == number)
    cout << "Correct!";
```

这里采用的缩进风格与前面建议的稍有不同。如果老老实实地遵循以前的缩进规则，那么似乎应该写成以下形式：

```
if (guess > number)
    cout << "Too high.";
else
    if (guess < number)
        cout << "Too low.";
    else
        if (guess == number)
            cout << "Correct!";
```

应该使用上一段代码的缩进模式，不要用这种

这证明了在少数情况下，不应该遵循嵌套语句的常规缩进规则。因为对齐所有 else 之后，可以同时对齐所有成对的“条件/行动”，使程序布局反映出你的期望。另一个原因是，如果 else 不停地右移，即便是不太深的 if-else 语句嵌套，也会很快跑到纸的外边。

由于条件是互斥的，所以在上述嵌套 if-else 语句中，最后一个 if 是多余的，可将

其省略。但有些时候，最好还是将它包括在一个注释中，如下所示：

```
if (guess > number)
    cout << "Too high.";
else if (guess < number)
    cout << "Too low.";
else // (guess == number)
    cout << "Correct!";
```

即使条件不互斥，也可使用这种形式的多分支 if-else 语句。无论条件是否互斥，计算机都会根据它们的出现顺序对条件进行求值。发现第一个为 true 的条件之后，就会采取与该条件对应的行动。没有任何条件为 true，就不采取任何行动。如果语句以一个没有配套 if 的 else 结尾，会在所有条件都为 false 时执行最后一个语句。

多路 if-else 语句

语法

```
if (Boolean_Expression_1)
    Statement 1
  else if (Boolean Expression 2)
    Statement 2
  …
  else if (Boolean Expression n)
    Statement n
  else
    Statement For All Other Possibilities
```

示例

```
if ((temperature < -10) && (day == SUNDAY))
    cout << "Stay home.";
else if (temperature < -10) // and day != SUNDAY
    cout << " Stay home, but call work.";
else if (temperature <= 0) // and temperature >= -10
    cout << "Dress warm.";
else // temperature > 0
    cout << "Work hard and play hard.";
```

布尔表达式按顺序求值，遇到第一个为 true 的布尔表达式，就执行相应的语句。没有任何布尔表达式为 true，就执行 Statement_For_All_Other_Possibilities。

编程实例　州收入税

图 3.5 的程序使用了一个多路 if-else 语句。该程序读取纳税人的净收入(取整为整数)，并根据这个净收入计算州收入税(state income tax)。假定这个州的计税方式如下。

(1) 15000 美元以下的净收入免税。

(2) 15001～25000 美元之间的每一美元净收入都征收 5%的税。

(3) 超过 25000 美元的每一美元净收入都征收 10%的税。

图 3.5 的程序使用了一个多路 if-else 语句，为上述三种情况分别采取一个行动。第二种情况的条件实际过于复杂了。除非已经尝试了第一个条件，并得到 false 的结论，否则是不会检查第二个条件的。所以，一旦计算机开始尝试第二个条件，就表示 netIncome 肯定大于 15000。所以，下面这行代码：

```
else if((netIncome > 15000) && (netIncome <= 25000))
```

可替换成下面的形式，两者结果相同：

```
else if (netIncome <= 25000)
```

图 3.5　多路 if-else 语句

```
1  // 该程序用于计算州收入税
2  #include <iostream>
3  using namespace std;
4
5
6  // 这个程序返回州收入税金额，计算方式如下：
7  // $15,000 以下的部分免税；$15,001～$25,000 之间税率为 5%；超过$25,000 税率为 10%
8
9  int main()
10 {
11     int netIncome;
12     double taxBill;
13     double fivePercentTax, tenPercentTax;
14
15
16     cout << "Enter net income (rounded to whole dollars) $";
17     cin >> netIncome;
18
19     if (netIncome <= 15000)
20         taxBill = 0;
21     else if ((netIncome > 15000) && (netIncome <= 25000))
22         // 为超过$15,000 的部分返回 5%的税金
23         taxBill = (0.05 * (netIncome-15000));
24     else // netIncome > $25,000
25     {
26         // fivePercentTax = $15,000～$25,000 之间的净收入的 5%
27         fivePercentTax = 0.05*10000;
28         // tenPercentTax = $25,000 以上净收入的 10%
29         tenPercentTax = 0.10 * (netIncome - 25000);
30         taxBill = (fivePercentTax + tenPercentTax);
31     }
32
33     cout.setf(ios::fixed);
34     cout.setf(ios::showpoint);
35     cout.precision(2);
36     cout  << "Net income=$" << netIncome << endl
37           << "Tax bill=$" << taxBill << endl;
38
39     return 0;
40 }
```

示范对话

```
Enter net income (rounded to whole dollars) $25100
Net income = $25100.00
Tax bill = $510.00
```

√ 自测题

5. 将以下代码嵌入一个完整的程序中，会得到什么输出结果？

```
int x = 2;
cout << "Start\n";
if (x <= 3)
    if (x != 0)
        cout << "Hello from the second if.\n";
    else
        cout << "Hello from the else.\n";
cout << "End\n";
cout << "Start again\n";
```

```
if (x > 3)
    if (x != 0)
        cout << "Hello from the second if.\n";
    else
        cout << "Hello from the else.\n";
cout << "End again\n";
```

6. 将以下代码嵌入一个完整的程序中，会得到什么输出结果？

```
int extra = 2;
if (extra < 0)
    cout << "small";
else if (extra == 0)
    cout << "medium";
else
    cout << "large";
```

7. 将自测题 6 的赋值语句改成下面这样，会得到什么输出结果？

```
int extra = -37;
```

8. 将自测题 6 的赋值语句改成下面这样，会得到什么输出结果？

```
int extra = 0;
```

9. 将以下代码嵌入一个完整的程序中，会得到什么输出结果？

```
int x = 200;
cout << "Start\n";
if (x < 100)
    cout << "First Output.\n";
else if (x > 10)
    cout << "Second Output.\n";
else
    cout << "Third Output.\n";
cout << "End\n";
```

10. 将自测题 9 的布尔表达式 (x > 10) 变成 (x > 100)，会得到什么输出结果？

11. 将以下代码嵌入一个完整的程序中，会得到什么输出结果？

```
int x = SOME_CONSTANT;
cout << "Start\n";
if (x < 100)
    cout << "First Output.\n";
else if (x > 100)
    cout << "Second Output.\n";
else
    cout << x << endl;
cout << "End\n";
```

SOME_CONSTANT 的值由你自己设置。它是 int 类型的常量。假定最后输出的既不是"First Output."，也不是"Second Output."，那么应该将 SOME_CONSTANT 设成哪个值是不言而喻的。

12. 写多路 if-else 语句，将 int 变量 n 的值划分为以下类别，并输出合适的消息：

n < 0 或者 0 ≤ n ≤ 100 或者 n > 100

13. 给定以下声明和输出语句，假定将它嵌入一个正确的程序中，会得到什么输出结果？

```
enum Direction { N, S, E, W };
// …
cout << W << " " << E << " " << S << " " << N << endl;
```

14. 给定以下声明和输出语句，假定将它嵌入一个正确的程序中，会得到什么输出结果？

```
enum Direction { N = 5, S = 7, E = 1, W };
// …
cout << W << " " << E << " " << S << " " << N << endl;
```

switch 语句

前面用 if-else 语句构造多路分支。**switch 语句**也能实现多路分支。图 3.6 展示了一个简单 switch 语句，它有 4 个标准分支，第 5 个分支为无效输入而设。变量 grade(成绩)决定了具体执行哪个分支。成绩'A'，'B'和'C'分别对应一个分支。成绩'D'和'F'执行同一个分支，不区别对待。如果 grade 的值是除了'A'，'B'，'C'，'D'和'F'之外的其他任何无效字符，就执行标识符 default 之后的 cout 语句。

图 3.6　switch 语句

```
1  // 该程序用于演示 switch 语句
2  #include <iostream>
3  using namespace std;

4  int main()
5  {

6      char grade;
7      cout << "Enter your midterm grade and press return: ";
8      cin >> grade;

9      switch (grade)
10     {
11         case 'A':
12             cout << "Excellent. "
13                  << "You need not take the final.\n";
14           break;
15         case 'B':
16             cout << "Very good. ";
17             grade = 'A';
18             cout << "Your midterm grade is now "
19                  << grade << endl;
20             break;
21         case 'C':
22             cout << "Passing.\n";
23             break;
24         case 'D':
25         case 'F':
26             cout << "Not good. "
27                  << "Go study.\n";
28             break;
29         default:
30             cout << "That is not a possible grade.\n";
31     }

32     cout << "End of program.\n";
33     return 0;
34  }
```

示范对话 1

```
Enter your midterm grade and press Return: A
Excellent. You need not take the final.
End of program.
```

示范对话 2

```
Enter your midterm grade and press Return: B
Very good. Your midterm grade is now A.
End of program.
```

示范对话 3

```
Enter your midterm grade and press Return: D
Not good. Go study.
End of program.
```

示范对话 4

```
Enter your midterm grade and press Return: E
That is not a possible grade.
End of program.
```

视频讲解：*switch statement example*

在图 3.6 的示范 switch 语句和后面的补充内容“switch 语句”中，我们展示了 switch 的语法以及首选的缩进模式。

执行 switch 语句时，会在多个分支中选择一个，具体由关键字 switch 之后的圆括号中的**控制表达式**决定。在图 3.6 的示范 switch 语句中，控制表达式具有 char 类型。switch 语句的控制表达式必须返回以下值之一：布尔值、enum 常量、某个整数类型或者一个字符。执行 switch 语句时，会对这个控制表达式进行求值。然后，计算机会检查每个 case 标识符之后给定的常量值。如果发现一个常量等于控制表达式的值，就执行那个 case 的代码。例如，假定表达式求值为'B'，就会找到以下 case，并执行该行之后的代码：

```
case 'B':
```

注意，常量之后要跟一个冒号。还要注意，两个 case 不能够有相同的常量值，否则会产生歧义。

break 语句由关键字 break 和一个分号构成。计算机执行 case 标签之后的语句时，它会一直执行下去，直至遇到 break 语句。遇到 break 语句后，switch 语句会终止。如果省略 break 语句，在执行了一个 case 的代码之后，计算机会继续执行下一个 case 的代码。

switch 语句

语法

```
switch (Controlling Expression)
{
    case Constant 1:
        statement Sequence 1
        break;
     case Constant 2:
        Statement Sequence 2
        break;
            ...
     case Constant n:
        Statement Sequence n
        break;
     default:
        Default Statement Sequence
  }
```

示例

```
int vehicleClass; // vehicleClass 代表车的种类
cout << "Enter vehicle class: ";
```

```
cin >> vehicleClass;

switch (vehicleClass)
{
    case 1:
        cout << "Passenger car.";
        toll = 0.50;
        break;      <-- 遗漏这个 break 语句，小客车(passenger car)就要支付 1.5 而不是 0.5 美元
    case 2:
        cout << "Bus.";
        toll = 1.50;
        break;
    case 3:
        cout << "Truck.";
        toll = 2.00;
        break;
    default:
        cout << "Unknown vehicle class!";
}
```

注意，可为同一个代码小节使用两个 case 标签。在图 3.6 所示的 switch 语句中，会为值'D'和'F'采取相同的行动。可利用这种技术同时支持字母的大写和小写形式。例如在图 3.6 的程序中，为了同时支持字母'a'和'A'，可将以下语句：

```
case 'A':
    cout << "Excellent. "
         << "You need not take the final.\n";
    break;
```

替换成以下形式：

```
case 'A':
case 'a':
    cout << "Excellent. "
         << "You need not take the final.\n";
    break;
```

当然，也可对其他所有字母进行同样的处理。

如果所有 case 标签都没有一个与控制表达式相匹配的常量，就执行 default 标签之后的语句。default 小节并不是必须的。如果没有 default 小节，控制表达式的值也没有匹配项，那么在执行了 switch 语句之后，什么事情都不会发生。不过，最保险的做法是总是包括一个 default 小节。如果认为自己的 case 标签已经覆盖了所有可能的情况，就可以在 default 小节显示一条错误消息。图 3.6 正是这样做的。

陷阱：忘记在 switch 语句中添加 break

如果忘记在 switch 语句中添加 break，编译器不会报错。虽然写的是语法正确的 switch 语句，但不一定能获得想要的效果。以补充内容“switch 语句”中的 switch 语句为例。如果遗漏 break 语句(箭头标注的那个)，那么在变量 vehicleClass 的值为 1 的情况下，虽然会正确执行有 case 1:标签的那个 case，但计算机还会继续执行下一个 case。这会产生令人迷惑的输出，因为它先说一辆车是小客车(passenger car)，接着又说它是巴士(bus)。此外，toll 的终值是 1.50，但本来应该是 0.50。计算机开始执行一个 case 语句后，除非遇到一个 break 语句或者到达 switch 语句的结尾，否则是不会停止的。 ■

为菜单使用 switch 语句

多路 if-else 语句用途比 switch 语句更广。所有使用 switch 语句的地方都能改为使用多路 if-else 语句。但 switch 语句有时更直观。例如，switch 语句尤其适合用来实现菜单。

饭店菜单列出了各种菜式供客人挑选。计算机程序的**菜单**是在屏幕上显示的供用户选择的列表，用户可从中选择不同的选项。图 3.7 的程序为学生提供了有关家庭作业的信息。程序用菜单允许学生选择自己希望的信息。实现菜单操作一个可读性更好的方式是使用函数。函数的详情将在第 4 章讨论。

图 3.7　一个菜单

```
1  // 该程序用于提供家庭作业信息
2  #include <iostream>
3  using namespace std;
4
5
6  int main()
7  {
8      int choice;
9
10     do
11     {
12         cout << endl
13              << "Choose 1 to see the next homework assignment.\n"
14              << "Choose 2 for your grade on the last assignment.\n"
15              << "Choose 3 for assignment hints.\n"
16              << "Choose 4 to exit this program.\n"
17              << "Enter your choice and press Return: ";
18         cin >> choice;
19
20         switch (choice)
21         {
22             case 1:
23                 // 这里的代码显示下一个作业
24                 break;
25             case 2:
26                 // 这里的代码要求提供学生编
27                 // 号并给出相应的成绩
28                 break;
29             case 3:
30                 // 这里的代码显示当前作业的
31                 // 提示
32                 break;
33             case 4:
34                 cout << "End of Program.\n";
35                 break;
36             default:
37                 cout << "Not a valid choice.\n"
38                      << "Choose again.\n";
39         }
40     } while (choice != 4);
41
42     return 0;
43 }
```

示范对话

```
Choose 1 to see the next homework assignment.
Choose 2 for your grade on the last assignment.
Choose 3 for assignment hints.
Choose 4 to exit this program.
Enter your choice and press Return: 3

Assignment hints:
Analyze the problem.
Write an algorithm in pseudocode.
Translate the pseudocode into a C++ program.

Choose 1 to see the next homework assignment.
Choose 2 for your grade on the last assignment.
Choose 3 for assignment hints.
Choose 4 to exit this program.
Enter your choice and press Return: 4
End of Program.
```

确切输出取决于插入 switch 语句的代码

代码块

switch 语句或者 if-else 语句的每个分支都是一个独立的子任务。最好将每个分支的操作都变成一个函数调用。这样一来，每个分支的子任务就可以单独设计、编写和测试。另一方面，某个分支的操作也许过于简单，所以只需把它变成一个复合语句。有时可以考虑为这个复合语句提供它自己的局部变量。以图 3.8 的程序为例，它为给定价格和数量的商品计算总价款。如果是一次批发交易，就不征收销售税(可能是因为要在商品转卖给零售顾客时征税)。但是，如果是一次零售交易，就必须加上销售税。一个 if-else 语句用于为批发和零售执行不同的计算。对于零售交易，计算时要用到一个名为 subtotal 的临时变量。所以，我们在与 if-else 语句的那个分支对应的复合语句中声明该变量。

图 3.8　有局部变量的代码块

```
1  // 计算批发或零售交易的总价款
2  #include <iostream>
3  using namespace std;
4
5
6  int main()
7  {
8      const double TAX_RATE = 0.05; // 5%的销售税
9      char saleType;
10     int number;
11     double price, total;
12
13     cout << "Enter price $";
14     cin >> price;
15     cout << "Enter number purchased: ";
16     cin >> number;
17     cout << "Type W if this is a wholesale purchase.\n"
18          << "Type R if this is a retail purchase.\n"
19          << "Then press return.\n";
20     cin >> saleType;
21
22     if ((saleType == 'W') || (saleType == 'w'))
23     {
24         total = price * number;
25     }
26     else if ((saleType == 'R') || (saleType == 'r'))
```

```
27      {
28          double subtotal;  ←——— 这个块的局部变量
29          subtotal = price * number;
30          total = subtotal + subtotal * TAX_RATE;
31      }
32      else
33      {
34          cout << "Error in input.\n";
35      }
36      cout.setf(ios::fixed);
37      cout.setf(ios::showpoint);
38      cout.precision(2);
39      cout << number << " items at $" << price << endl;
40      cout << "Total Bill = $" << total;
41      if ((saleType == 'R') || (saleType == 'r'))
42          cout << " including sales tax.\n";
43
44      return 0;
45  }
```

示范对话

```
Enter price: $10.00
Enter number purchased: 2
Type W if this is a wholesale purchase.
Type R if this is a retail purchase.
Then press Return.
R
2 items at $10.00
Total Bill = $21.00 including sales tax.
```

如图 3.8 所示，变量 subtotal 在复合语句中声明。如果愿意，subtotal 这个变量名可以在复合语句的外部用于其他用途，不会和内部的变量冲突。变量在复合语句内部声明，就说该变量**局部**于这个复合语句。局部变量在复合语句执行时创建，在复合语句结束时销毁。换言之，局部变量只存在于声明它的复合语句内部。在复合语句内部，除了能使用复合语句中声明的局部变量，还能使用在复合语句外部声明的所有变量。

包含变量声明的复合语句不是简单的复合语句，它有一个特殊名称。通常将包含变量声明的复合语句称为**块**。块内声明的变量称为这个块的局部变量，或者说它们具有块的作用域(不包含任何变量声明的复合语句也可称为块。用花括号封闭的所有代码统称为一个块)。

语　句　块

代码块是用花括号封闭起来的 C++代码。块内声明的变量是那个块的局部变量，所以变量名能在块的外部用于其他用途(比如由一个不同的变量重用该名称)。

第 4 章将介绍如何定义函数。函数定义的主体也是一个块。对于非函数主体的块，没有一个标准的名称来称呼它们。然而，我们仍然有必要讨论这些块，所以事先需要约定好一个名称。如果一个块不是函数的主体(也不是程序的 main 部分的主体)，我们就把它称为代码块。

代码块可嵌套于其他代码块中，刚才说的局部变量规则同样适用于这些嵌套的代码块。但在代码块嵌套的情况下，运用那些规则会增大程序的复杂性。一个更好的规则是根本不要嵌套代码块。嵌套的代码块使程序难以阅读。如果认为确实有必要嵌套代码块，请将部分代码块转变成函数定义，并用函数调用代替嵌套代码块。事实上，任何类型的代码块都

不能滥用。大多数时候，函数调用都要优于代码块。为保持内容的完整，我们在下面的补充内容中总结了嵌套块的作用域规则。

> **嵌套块的作用域规则**
>
> 假定一个块嵌套于另一个块中，而且两个块都声明了同名变量，就得到了同名但不相同的两个变量。一个变量只存在于内层的块中，不可在它的外部访问。另一个变量只存在于外部块中，不可在内部块中访问。这是两个不同的变量，修改任何一个都不影响另一个。

陷阱：疏忽局部变量

在一对花括号中声明变量时，此变量会成为由花括号封闭的那个块的局部变量。无论你的原意是不是让这个变量成为局部变量，结果都是如此。如果希望变量能在花括号的外部使用，就必须在花括号的外部声明它。■

√ 自测题

15. 将以下代码嵌入一个完整的程序中，会得到什么输出结果？

```
int firstChoice = 1;
switch (firstChoice + 1)
{
    case 1:
        cout << "Roast beef\n";
        break;
    case 2:
        cout << "Roast worms\n";
        break;
    case 3:
        cout << "Chocolate ice cream\n";
    case 4:
        cout << "Onion ice cream\n";
        break;
    default:
        cout << "Bon appetit!\n";
}
```

16. 将自测题 15 的第一行变成下面这样，会得到什么输出结果？

```
int firstChoice = 3;
```

17. 将自测题 15 的第一行变成下面这样，会得到什么输出结果？

```
int firstChoice = 2;
```

18. 将自测题 15 的第一行变成下面这样，会得到什么输出结果？

```
int firstChoice = 4;
```

19. 将以下代码嵌入一个完整的程序中，会得到什么输出结果？

```
int number = 22;
{
    int number = 42;
    cout << number << " ";
}
cout << number;
```

20. 虽然不建议采用这种风格来编程，但在这个练习中，我们故意使用嵌套块来帮助你理解作用域规则。将以下代码嵌入一个完整的程序中，会得到什么输出结果？

```
{
    int x = 1;
    cout << x << endl;
    {
        cout << x << endl;
        int x = 2;
        cout << x << endl;
        {
            cout << x << endl;
            int x = 3;
            cout << x << endl;
        }
        cout << x << endl;
    }
    cout << x << endl;
}
```

3.3　C++循环语句详解

人生无非是一件又一件的倒霉事。

——德娜·文森特·默蕾，写给亚瑟·戴瑞森·菲克的信，1930 年 10 月 24 日

任何能多次重复一个或一组语句的程序结构都是**循环**。我们学过的简单 while 和 do-while 语句就是循环。在循环中重复的语句(或语句组)称为**循环主体**或**循环体**。循环主体的每一次重复都称为一次**循环迭代**。构造循环时主要考虑两个设计问题：循环主体是什么？循环主体要迭代多少次？

while 语句回顾

图 3.9 回顾了 while 语句及其变体 do-while 语句的语法。两者的关键区别在于**什么时候**检查控制布尔表达式。执行 while 语句时，布尔表达式在循环主体执行**之前**检查。如果布尔表达式一开始就求值为 false，主体根本不会执行。执行 do-while 语句时，先执行循环主体，在循环主体执行*之后*检查布尔表达式。因此，do-while 语句至少执行循环主体一次。在此之后，while 循环和 do-while 循环以非常相似的方式工作。循环主体每一次迭代之后，都会再次检查布尔表达式；为 true，就再次迭代。如果从 true 变成了 false，循环终止。

图 3.9　while 语句和 do-while 语句的语法

以单个语句作为主体的 while 语句

```
while (Boolean_Expression)
    Statement              <------ 主体
```

以多个语句作为主体的 while 语句

```
while (Boolean_Expression)
{
    Statement_1       <--+
    Statement_2          |--主体
        ⋮                |
    Statement_Last    <--+
}
```

以单个语句作为主体的 do-while 语句

```
do
    Statement  ←———— 主体
while (Boolean_Expression);
```

以多个语句作为主体的 do-while 语句

```
do
{
    Statement_1     ←—┐
    Statement_2       ├—— 主体
      ⋮               │
    Statement_Last  ←—┘
} while (Boolean_Expression);
```

---

while 循环执行时，发生的第一件事是对控制布尔表达式求值。如布尔表达式当时就求值为 false，循环主体一次都不会执行。虽然循环主体执行 0 次看起来没意义，但有时正需如此。例如，经常要用 while 循环累加一组数字，但数字列表可能为空。更具体的一个例子，支票簿结余程序可能用一个 while 循环累加你在一个月中开具的所有支票的金额。但是，你可能花了一个月的时间去旅游，没有开具任何支票。这种情况下，需要累加的就是 0 个数字，所以循环应该迭代 0 次。

再论递增操作符和递减操作符

以前一直将递增操作符作为语句来使用，目的是使变量值增 1。例如，以下语句在屏幕上输出 42：

```
int number = 41;
number++;
cout << number;
```

但递增操作符是真正的操作符，像 number++这样的表达式会返回值，所以可以在如下所示的表达式中使用 number++：

```
2 * (number++)
```

表达式 number++首先返回变量 number 的值，然后使 number 的值递增 1。例如：

```
int number = 2;
int valueProduced =2 * (number++);
cout << valueProduced << endl;
cout << number << endl;
```

输出结果如下：

```
4
3
```

请注意表达式 2 * (number++)。C++对这个表达式进行求值时，使用的是 number 递增前的值，而不是递增后的值。因此，表达式 number++生成的值是 2，虽然递增操作符最后还是将 number 的值变成了 3。这表面上很怪，但有时正是你所希望的。另外，就像下面讲到的，如果希望表达式有不同的行为，也能如愿以偿。

表达式 v++首先求值为变量 v 的值，然后变量 v 的值递增 1。相反，如果将++放到变

量名之前，上述两个操作的顺序也会反过来。例如，表达式++v 先使变量 v 的值递增 1，再返回 v 递增过的值。例如：

```
int number = 2;
int valueProduced =2 * (++number);
cout << valueProduced << endl;
cout << number << endl;
```

输出结果如下：

```
6
3
```

注意，两个递增操作符 number++和++number 对于变量 number 来说具有相同的效果，都使 number 的值递增 1。但两个表达式产生了不同的求值结果。记住，++在变量之前，先递增再返回值；++在变量之后，先返回值再递增。

图 3.10 的程序在 while 循环中使用递增操作符来统计循环主体的重复次数。递增操作符的主要用途之一就是控制循环迭代，图 3.10 对此进行了清楚的说明。

图 3.10　递增操作符作为表达式使用

```
1  // 卡路里统计程序
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7     int  numberOfItems, count,
8          caloriesForItem, totalCalories;
9
10    cout << "How many items did you eat today? ";
11    cin >> numberOfItems;
12
13    totalCalories = 0;
14    count = 1;
15    cout << "Enter the number of calories in each of the\n"
16         << numberOfItems << " items eaten:\n";
17
18    while (count++ <= numberOfItems)
19    {
20       cin >> caloriesForItem;
21       totalCalories =  totalCalories
22                        + caloriesForItem;
23    }
24
25    cout << "Total calories eaten today = "
26         << totalCalories << endl;
27    return 0;
28 }
29
```

示范对话

```
How many items did you eat today?
7
Enter the number of calories in each of the
7 items eaten:
300 60 1200 600 150 1 120
Total calories eaten today = 2431
```

关于递增操作符的一切同样适用于递减操作符，只是变量值递减 1，而不是递增 1。例如以下代码：

```
int number = 8;
int valueProduced = number--;
cout << valueProduced << endl;
cout << number << endl;
```

输出结果如下：

```
8
7
```

另一方面，以下代码：

```
int number = 8;
int valueProduced = --number;
    cout << valueProduced << endl;
    cout << number << endl;
```

输出结果如下：

```
7
7
```

number--首先返回 number 的值，再使 number 递减；另一方面，--number 先使 number 递减，再返回 number 的值。

不可以将递增与递减操作符应用于除单个变量以外的其他任何东西。在 C++中，(x + y)++，--(x + y)和 5++等表达式都是无效的。

√ 自测题

21. 将以下代码嵌入一个完整的程序中，会得到什么输出结果？

```
int count = 3;
while (count-- > 0)
   cout << count << " ";
```

22. 将以下代码嵌入一个完整的程序中，会得到什么输出结果？

```
int count = 3;
while (--count > 0)
     cout << count << " ";
```

23. 将以下代码嵌入一个完整的程序中，会得到什么输出结果？

```
int n = 1;
do
    cout << n << " ";
while (n++ <= 3);
```

24. 将以下代码嵌入一个完整的程序中，会得到什么输出结果？

```
int n = 1;
do
    cout << n << " ";
while (++n <= 3);
```

for 语句

while 和 do-while 语句是绝对需要的循环机制。事实上，仅 while 语句便已足够。然而，有一种循环过于普遍，以至于 C++专门为它提供了一个语句。执行数值计算时，经

常要用数字 1 执行一个计算，再用数字 2，再用数字 3，依此类推，直到抵达某个终值。例如，为了累加 1~10 的整数，你希望计算机执行以下语句 10 次。第一次 n 的值等于 1，以后每次 n 值递增 1：

```
sum = sum + n;
```

下面是用 while 语句来实现它的一种方式：

```
sum = 0;
n = 1;
while (n <= 10)
{
    sum = sum + n;
    n++;
}
```

虽然 while 循环确实可行，但这种情况下最适合采用 **for 语句**(也称为 for 循环)。以下 for 语句完成相同的任务：

```
sum = 0;
for (n = 1; n <= 10; n++)
    sum = sum + n;
```

下面将分析 for 语句的组成部分。

首先注意，while 循环和 for 循环由相同的部分组合而成。最开始都是一个赋值语句，将变量 sum 设为 0。在两种情况下，对 sum 的赋值都在循环语句开始之前完成。循环本身都由以下几个部分构成：

```
n = 1; n <= 10; n++; 和 sum = sum + n;
```

这些部分在 for 语句和 while 语句中的功能是一样的。for 语句只是表示同一件事情的更精简的方式。虽然还存在其他方案，但我们以后只用 for 语句执行由一个变量控制的循环。上例的控制变量是 n。了解两种循环的共同点之后，接着了解 for 语句的编写规则。

for 语句以关键字 for 开头，随后要在圆括号中指定三点，告诉计算机如何处理控制变量。for 语句的头部看起来像下面这样：

```
for (Initialization_Action; Boolean_Expression; Update_Action)
```

第一个表达式指出变量如何初始化；第二个是布尔表达式，用于检查循环应在什么时候终止；最后一个表达式指出循环主体每次迭代之后，循环控制变量如何更新。在前面的例子中，for 循环头如下：

```
for (n = 1; n <= 10; n++)
```

n = 1 指出 n 初始化为 1。n <= 10 指出只要 n 小于或等于 10，循环主体就继续迭代。最后一个表达式 n++指出每次执行了循环主体之后，n 值递增 1。

for 循环头的三个表达式要用(而且只能用)两个分号来分隔。绝对不要在第三个表达式之后添加分号(从技术上说，这三样东西是表达式而不是语句，所以结尾处不需要分号)。

图 3.11 展示了 for 语句的语法，并将 for 语句转换成一个等价的 while 语句，从而描述了它具体采取的操作。注意，在 for 语句中，和在相应的 while 语句中一样，循环结束条件是在第一次循环迭代之前测试的。因此，for 循环主体可能执行 0 次。

图 3.11　for 语句

for 语句

语法

```
1  for (Initialization_Action; Boolean_Expression; Update_Action)
2       Body_Statement
```

示例

```
1  for (number = 100; number >= 0; number--)
2      cout << number
3           << " bottles of beer on the shelf.\n";
```

等价的 while 循环

等价语法

```
1  Initialization_Action;
2  while (Boolean_Expression)
3  {
4      Body_Statement;
5      Update_Action;
6  }
```

等价示例

```
1  number = 100;
2  while (number >= 0)
3  {
4      cout << number
5           << " bottles of beer on the shelf.\n";
6      number--;
7  }
```

输出

```
100 bottles of beer on the shelf.
99 bottles of beer on the shelf.
.
.
.
0 bottles of beer on the shelf.
```

图 3.12 展示了已嵌入完整程序(虽然这个程序非常简单)的示范 for 语句。图 3.12 的 for 语句与前面讨论的相似，区别在于变量 n 是在初始化为 1 时声明的。换言之，n 是在 for 语句内部声明的。for 语句的初始化操作可包含变量声明。如果变量只在 for 语句中使用，for 语句头就是声明变量的最佳场所。然而，如果还要在 for 语句外部使用，最好在 for 语句外部声明。

图 3.12　一个 for 语句

```
1   // 演示 for 循环
2   #include <iostream>
3   using namespace std;
4
5   int main()
6   {
7      int sum = 0;
8
9      for (int n = 1; n <= 10; n++)  // 注意，n 是 for 循环体的局部变量
10        sum = sum + n;
11
12     cout << "The sum of the numbers 1 to 10 is "
```

初始化动作　只要为 true，就重复循环　在循环主体每次迭代后求值

```
13          << sum << endl;
14      return 0;
15  }
```

输出

```
The sum of the numbers 1 to 10 is 55
```

ANSI C++标准规定，对于 for 循环头中的任何声明，都假定它相对于循环主体来说是“局部”的。但早期 C++编译器没有遵守这个规定。应确定自己的编译器如何对待 for 循环头中声明的变量。考虑到移植性，不应编写依赖于这种行为的代码。

我们对 for 语句的描述还是显得过于具体。for 循环头的三个表达式可以是任何 C++表达式，所以涉及的变量数可能多于(甚至少于)一个。但本书只用一个。

在图 3.12 的 for 语句中，主体是一个简单的赋值语句：

```
sum = sum + n;
```

主体可为任何语句。可以是复合语句，允许多个语句构成 for 循环体，如图 3.13 所示。

图 3.13　多个语句构成的 for 循环体

语法

```
for (Initialization_Action; Boolean_Expression; Update_Action)
{
    Statement_1      <--┐
    Statement_2         |
        .               |
        .               ├── 主体
        .               |
    Statement_Last   <--┘
}
```

示例

```
for (int number = 100; number >= 0; number--)
{
    cout << number
         << " bottles of beer on the shelf.\n";
    if (number > 0)
         cout << "Take one down and pass it around.\n";
}
```

到目前为止，每次 for 循环迭代，控制变量要么递增 1，要么递减 1。但还有其他变量更新方式。变量可递增或递减 2 或 3 (或任何数字)。如变量是 double 类型，甚至能递增或递减一个小数。以下 for 循环都是合法的：

```
int n;
for (n = 1; n <= 10; n = n + 2)
    cout << "n is now equal to " << n << endl;

for (n = 0; n > -100; n = n - 7)
    cout << "n is now equal to " << n << endl;

for (double size = 0.75; size <= 5; size = size + 0.05)
    cout << "size is now equal to " << size << endl;
```

甚至不一定要通过增减来更新。另外，初始化时不一定要让变量等于一个常量。可以采取自己喜欢的任何方式初始化和修改循环控制变量。例如，以下语句展示了另一种开始 for 循环的方式：

```
for (double x = pow(y, 3.0); x > 2.0; x = sqrt(x))
     cout << "x is now equal to " << x << endl;
```

陷阱：for 语句中多余的分号

不要在 for 循环头的结束圆括号之后添加分号。为体会这一点，请看以下 for 循环：

```
for (int count = 1; count <= 10; count++);   ←—— 分号问题
    cout << "Hello\n";
```

如果没有注意到那个多余的分号，可能以为该 for 循环会在屏幕上写 10 个 Hello。注意到分号，可能以为编译器会报告一条错误消息。但这两件事情都不会发生。在一个完整的程序中嵌入上述 for 循环，编译器根本不报错。但它只在屏幕上输出一个 Hello，而不是 10 个。为什么？为了理解这个问题，还需要学习一点背景知识。

在 C++中创建语句的一种方式是在代码之后放一个分号。在 x++之后放一个分号，就会将以下表达式：

```
x++
```

变成以下语句：

```
x++;
```

如独立使用分号，也会创建一个语句。也就是说，分号本身就是一个语句，它称为**空语句**，也称为 **null 语句**。空语句不执行任何操作，但仍是一个语句。所以，下面是一个完整和合法的 for 循环，它的主体就是一个空语句：

```
for (int count = 1; count <= 10; count++);
```

以上 for 循环确实会迭代 10 次，但由于主体是空语句，所以主体虽然会迭代 10 次，但每次都不采取任何行动。

现在重新审视一下前面那个有“分号问题”的 for 循环代码。由于含有额外分号，所以 for 循环具有一个空主体。我们现在知道，那个 for 循环什么事情都不会做。for 循环结束之后，会执行以下 cout 语句，并将 Hello 写到屏幕：

```
cout << "Hello\n";
```

具有空主体的 for 循环也有一定的用处，这种情况以后可能碰到。但就现阶段来说，这样的 for 循环通常是一种因为疏忽而犯的**逻辑**错误。　■

应该使用哪种循环

设计循环时，具体要用哪种 C++循环语句最好推迟到设计阶段的末期再做决定。首先用伪代码设计出循环，再将伪代码转换成 C++代码。转换时，很容易决定具体使用什么 C++循环语句。

如果循环要求进行数值计算，而且要用到一个每次循环都发生均匀变化的变量，就使用 for 循环。事实上，但凡要用循环执行数值计算，就应考虑 for。虽然并非肯定适合，但在执行数值计算时，for 通常都是最清楚、最容易实现的循环。

其他大多数情况都应使用 while 或 do-while 循环；很容易决定具体选用哪一个。如果希望循环主体至少执行一次，就用 do-while。如果循环主体可能一次都不执行，就必须使用 while。使用 while 循环的一个常见的情形是：虽然要读取输入，但允许没有任何数据。例如，假定程序要为学生读取所有科目的考试分数，但学生可能没有参加任何一门考试。由于输入循环可能遇到一个空列表，所以需要使用 while 循环。

√ 自测题

25. 将以下代码嵌入一个完整的程序中，会得到什么输出结果？

```
for (int count = 1; count < 5; count++)
    cout << (2 * count) << "";
```

26. 将以下代码嵌入一个完整的程序中，会得到什么输出结果？

```
for (int n = 10; n > 0; n = n - 2)
{
   cout << "Hello ";
   cout << n << endl;
}
```

27. 将以下代码嵌入一个完整的程序中，会得到什么输出结果？

```
for (double sample = 2; sample > 0; sample = sample - 0.5)
     cout << sample << " ";
```

28. 在以下每种情况中，最适合使用哪种循环(while，do-while 或者 for)：

a. 数列累加，比如 1/2 + 1/3 + 1/4 + 1/5 + ⋯ + 1/10。
b. 读入学生的所有科目的考试分数列表。
c. 读取某部门员工的病假天数。
d. 测试一个函数，检查向其传递不同参数值时的结果。

29. 将以下循环改写为 for 循环：

a.
```
int i = 1;
while ( i <= 10)
{
    if (i < 5 && i != 2)
        cout << 'X';
    i++;
}
```

b.
```
int i = 1;
while ( i <= 10)
{
    cout << 'X';
    i = i + 3;
}
```

c.
```
long m = 100;
do
{
    cout << 'X';
    m = m + 100;
} while (m < 1000);
```

30. 以下循环会输出什么结果？说明 n 值和变量 log 的值的关系。

```
int n = 1024;
int log = 0;
for (int i = 1; i < n; i = i * 2)
    log++;
cout << n << " " << log << endl;
```

31. 以下循环会输出什么结果？然后再对这段代码进行评述。

```
int n = 1024;
int log = 0;
for (int i = 1; i < n; i = i * 2);
    log++;
cout << n << " " << log << endl;
```

32. 以下循环会输出什么结果？然后再对这段代码进行评述。

```
int n = 1024;
int log = 0;
for (int i = 0; i < n; i = i * 2)
   log++;
cout << n << " " << log << endl;
```

陷阱：未初始化的变量和无限循环

第 2 章首次介绍简单 while 和 do-while 循环时，我们已发出警告，要注意与循环有关的两个陷阱。我们指出，执行循环前，循环中要用到的所有变量都应该初始化(也就是要为其赋值)。说起来容易，但在实际设计循环时，许多人都会忘记在循环开始之前初始化变量。我们还指出，应该尽量避免无限循环(即死循环)。对于 for 循环，同样要注意这两个陷阱。 ■

break 语句

以前曾用 break 语句终止 switch 语句。break 语句还可用于退出循环。有时需要提前退出循环。例如，可能要在循环中检查不正确的输入，一旦遇到不正确的输入，就终止循环。图 3.14 的代码读取一组负数，将它们的值累加到变量 sum 中。用户连续输入 10 个负数，循环正常终止。但是，如果用户中途忘记为一个数字输入负号，就放弃计算，立即执行 break 语句以终止循环。

图 3.14　在循环中使用 break 语句

```
1  // 累加 10 个负数
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7      int number, sum = 0, count = 0;
8      cout << "Enter 10 negative numbers:\n";
9
10     while (++count <= 10)
11     {
12         cin >> number;
13
14         if (number >= 0)
15         {
16             cout << "ERROR: positive number"
17                  << " or zero was entered as the\n"
18                  << count << "th number! Input ends "
19                  << "with the " << count << "th number.\n"
20                  << count << "th number was not added in.\n";
21             break;
22         }
23
24         sum = sum + number;
25     }
26
27     cout << sum << " is the sum of the first "
28          << (count - 1) << " numbers.\n";
29
30     return 0;
31 }
```

示范对话

```
Enter 10 negative numbers:
-1 -2 -3 4 -5 -6 -7 -8 -9 -10
ERROR: positive number or zero was entered as the
4th number! Input ends with the 4th number.
4th number was not added in.
-6 is the sum of the first 3 numbers.
```

break 语句

break 语句可用于退出循环语句。执行 break 语句后，循环语句立即终止，并继续执行循环语句之后的语句。break 语句可用于任何形式的循环中，包括 while，do-while 或者 for 循环。我们已在 switch 语句中使用了同样的 break 语句。

陷阱：嵌套循环中的 break 语句

break 语句只终止包含它的最内层的循环。在循环中嵌套循环，而且在内层循环中使用了 break 语句，break 语句终止的只是内层循环。 ■

√ 自测题

33. 将以下代码嵌入一个完整的程序中，会得到什么输出结果？

```
int n = 5;
while (--n > 0)
{
    if (n == 2)
        break;
    cout << n << " ";
}
cout << "End of Loop.";
```

34. 将以下代码嵌入一个完整的程序中，会得到什么输出结果？

```
int n = 5;
while (--n > 0)
{
    if (n == 2)
        exit(0);
    cout << n << " ";
}
cout << "End of Loop.";
```

35. 解释 break 语句的作用。在哪些地方使用 break 语句是有效的？

3.4 设 计 循 环

转呀转，转呀转，她转着圈儿。她要停在哪儿，谁也不知道。

——*嘉年华上招揽客人时喊的话*

设计循环时要设计三个要素。

(1) 循环体。

(2) 初始化语句。

(3) 循环结束条件。

首先讲解两个常规的循环任务，并展示如何针对每个任务设计这三个要素。

求和与求乘积的循环

许多常规任务都要求读取一组数字并计算它们的和。如果事先知道有多少个数，可以用以下伪代码简单地完成该任务。变量 thisMany 的值就是要累加的数字个数。总和累加到变量 sum 中。

```
sum = 0;
重复以下代码 thisMany 次:
    cin >> next;
    sum = sum + next;
循环终止
```

上述伪代码很容易用 for 循环实现：

```
int sum = 0;
for (int count = 1; count <= thisMany; count++)
{
    cin >> next;
    sum = sum + next;
}
```

注意，执行以下循环语句主体之前，sum 必须赋值：

```
sum = sum + next;
```

由于首次执行该语句之前 sum 必须有一个值，所以 sum 必须在循环执行前初始化好。为了确定 sum 的初始值，请考虑一次循环迭代之后想发生什么。加了第一个数字后，sum 应该是那个数字。也就是说，首次循环后，sum + next 的值应该等于 next。所以很明显，sum 的值要初始化为 0。

求一组数字的乘积时，可采取与求和相似的方式。如以下代码所示：

```
int product = 1;
for (int count = 1; count <= thisMany; count++)
{
    cin >> next;
    product = product * next;
}
```

变量 product 必须有一个初始值。不要以为所有变量都应该初始化成 0。如果 product 初始化成 0，上述循环结束后，它仍然是 0。就像前面的 C++代码展示的那样，product 正确的初始值是 1，因为首次循环之后，我们希望 product 等于读入的第一个数字。

重复这么多(ThisMany)次

可用 for 循环使主体重复确定的次数。

伪代码

重复下面的代码 *ThisMany* 次：

Loop_Body

等价的 for 语句

```
for (int count = 1; count <= ThisMany; count++)
    Loop_Body
```

> 示例
>
> ```
> for (int count = 1; count <= 3; count++)
>     cout << "Hip, Hip, Hurry\n";
> ```

终止循环

有 4 种常用的方法终止输入循环，我们按以下顺序讨论。

(1) 已知长度的列表。

(2) 每次迭代都询问。

(3) 以哨兵值结束的列表。

(4) 判断是否用完所有输入。

如程序能事先确定输入列表的长度，无论是通过询问用户来确定，还是用其他方法确定，都可使用“重复 *n* 次”循环来准确读取 *n* 次输入。该方法称为**已知长度的列表**。

终止输入循环的第二种方法是在每次迭代之后都询问用户是否需要再次迭代？例如：

```
sum = 0;
cout << "Are there any numbers in the list? (Type\n"
     << "Y and Return for Yes, N and Return for No): ";
char ans;
cin >> ans;
while ((ans == 'Y') || (ans == 'y'))
{
    cout << "Enter number: ";
    cin >> number;
    sum = sum + number;
    cout << "Are there any more numbers? (Type\n"
         << "Y for Yes, N for No. End with Return.): ";
    cin >> ans;
}
```

但在读取一个长列表时，这就显得过于繁琐。假定以这种方式输入 100 个数字，用户的心情可能经历从高兴到嘲笑，从生气到麻木的转变。读取长列表时，最好只设计一个停止信号，这就是下面要讨论的方法。

用循环来读取用户通过键盘输入的一组值时，最好的方式或许是使用**哨兵值**。哨兵值是一个明显有别于所有正式值的值，这样才能利用它的特殊性来指示列表的终止。例如，假定循环要读取一组正数，就可将一个负数用作哨兵值来终止循环。例如，使用以下循环可累加一组非负的整数：

```
cout << "Enter a list of nonnegative integers.\n"
     << "Place a negative integer after the list.\n";
sum = 0;
cin >> number;
while (number >= 0)
{
    sum = sum + number;
    cin >> number;
}
```

注意，列表中的最后一个数字虽然会读入，但不会累加到 sum 中。为了求数字 1，2 和 3 的和，用户可在列表末尾添加一个负数，比如：

```
1 2 3 -1
```

最后一个-1 虽然会读入，但不会被累加。

为了像这样使用哨兵值，必须保证对于当前处理的数据类型来说，至少有一个值不会出现在输入值列表中，这样才能把它用作哨兵值。如果列表由任意整数构成，就没有哨兵值的位置。在这种情况下，需要用其他方法终止循环。

从文件读取输入时，也可使用哨兵值，但更常见的一种方法是检查是否读取了文件中的全部输入。如果是，就终止循环。6.2.4 节将讨论如何用这种方法终止输入循环。

前面讨论的用于终止输入循环的所有技术都属于更常规的技术的特例。常规技术包括：

- 由计数控制的循环；
- 迭代之前询问；
- 遇到标志条件时退出。

由计数控制的循环是指在循环开始之前便确定迭代次数，然后刚好迭代那么多次。前面讨论的“已知长度的列表”就是这种常规循环的特例。所有“重复这么多次”循环都属于由计数控制的循环。

前面已讨论了“迭代之前询问”技术。虽然也可把它用于除了输入循环之外的其他循环，但它最常见的用途就是处理输入。

前面还讨论了用哨兵值终止输入循环的技术。在例子中，程序将非负的整数读入 number 变量。一旦 number 接收到负数，就表明输入结束；那个负数就是哨兵值。这属于“遇到标志条件时退出”这种常规技术的特例。如果变量值发生改变，指出发生了特定的事件，就通常将该变量称为**标志**。在前面的示范输入循环中，标志就是变量 number；一旦它变成负值，就表明输入列表结束。

检查是否用完所有输入，从而结束一个文件输入，这是“遇到标志条件时退出”技术的另一个特例。在这种情况下，标志条件由系统确定。从文件读取输入时，系统要不断地检查是否抵达文件尾。

标志也可用于终止非输入循环。例如，下面这个示例循环查找可能成为助教(tutor)的高分学生。学生从 1 开始编号。循环将检查每个学生编号，判断学生是否获得一个高分。一旦发现高分学生，就终止循环。对于这个例子，设定 90 分或者 90 分以上的成绩为高分。代码 computeGrade(n) 调用一个用户自定义函数。在这个例子中，函数执行代码来计算学生 n 的成绩，返回 0 到 100 的值。这个值拷贝到变量 grade 中。第 4 章将详细讨论函数。

```
int n = 1;
grade = computeGrade(n);
while (grade < 90)
{
    n++;
    grade = computeGrade(n);
}
cout << "Student number " << n << " may be a tutor.\n"
     << "This student has a score of " << grade << endl;
```

在这个例子中，变量 grade 作为标志使用。

以上循环还展示了设计循环时可能遇到的一个问题。如果所有学生的分数都无法达到 90，那会发生什么？答案取决于函数 computeGrade 的定义。如果 n 可以为所有正整数，

就会产生一个无限循环。更糟的是，对于没有对应学生的参数 n，如果将 grade 定义成 100，就会将不存在的学生变成助教。在任何情况下，都有可能出错。假如循环可能变成一个无限循环，或者迭代次数超过实际需要的次数，就应检查循环是否迭代了太多的次数。例如，下面展示了上述循环的一个更好的条件，其中变量 numberOfStudents 等于学生数量。

```
int n = 1;
grade = computeGrade(n);
while (( grade < 90) && ( n < numberOfStudents))
{
    n++;
    grade = computeGrade(n);
}
if (grade >= 90)
    cout << "Student number " << n << " may be a tutor.\n"
         << "This student has a score of " << grade << endl;
else
    cout << "No student has a high score.";
```

嵌套循环

视频讲解：*Nested Loop Example*

图 3.15 的程序用于跟踪绿颈秃鹰(一个濒临灭绝的物种)的繁殖速度。在这种秃鹰生活的地区，动物保护学者每年都要统计绿颈秃鹰的产卵数量。图 3.15 的程序用于生成动物保护报告，统计所有人观察到的鸟蛋总数。

每个人的报表都由一组数字构成。每个数字都代表在其中一个绿颈秃鹰巢中发现的鸟蛋数量。我们希望每个人的报告都用一个 while 循环来读取，并计算他发现的鸟蛋总数。每个人输入的数字列表都以一个负数结尾，这个负数是哨兵值。一旦检测到哨兵值，就结束当前 while 循环，并继续读取下一个人的报告。

循环主体可包含任何语句，所以一个循环能嵌套到另一个循环中。图 3.15 的程序演示了循环如何嵌套。针对 count 的每个值(从 1 到 numberOfReports)，嵌套的循环都会执行一次。换言之，外层 for 循环每一次迭代，内层 while 循环都会完整执行一次。第 4 章将用函数增强该程序的可读性。

图 3.15　显式嵌套的循环

```
1  // 计算绿颈秃鹰产卵总数,
2  // 由保护区的所有动物保护学者统计
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8       cout << "This program tallies conservationist reports\n"
9       << "on the green-necked vulture.\n"
10      << "Each conservationist's report consists of\n"
11      << "a list of numbers. Each number is the count of\n"
12      << "the eggs observed in one"
13      << "green-necked vulture nest.\n"
14      << "This program then tallies
15      << "the total number of eggs.\n"
16
17     int numberOfReports;
18     cout << "How many conservationist reports are there? ";
19     cin >> numberOfReports;
20
21     int grandTotal = 0, subtotal, count;
```

```
22   for (count = 1; count <= numberOfReports; count++)
23   {
24       cout << endl << "Enter the report of "
25            << "conservationist number " << count << endl;
26       cout << "Enter the number of eggs in each nest.\n"
27            << "Place a negative integer at the end of your list.\n";"
28       subtotal = 0;
29       int next;
30       cin >> next;
31       while (next >=0)
32       {
33           subtotal = subtotal + next;
34           cin >> next;
35       }
36       cout << "Total egg count for conservationist "
37            << "number" << count << "is"
38            << subtotal << endl;
39       grandTotal = grandTotal + subtotal;
40   }
41
42   cout << endl << "Total egg count for all reports = "
43        << grandTotal << endl;
44
45   return 0;
46 }
```

√ 自测题

36. 写一个循环，嵌入一个完整的程序中之后，它应该在屏幕上写 10 个单词 Hello。

37. 写一个循环，它能读取一个偶数列表(比如 2，24，8，6)，并计算列表中的所有数字之和。列表以一个哨兵值结束。你要自己决定最适宜的哨兵值。

38. 预测以下嵌套循环的输出结果：

```
int n, m;
for (n = 1; n <= 10; n++)
    for (m = 10; m >= 1; m--)
        cout << n << " times " << m
             << " = " << n * m << endl;
```

调试循环

无论程序设计得多细心，错误在所难免。就循环来说，程序员经常犯某些固定的错误。大多数循环错误都涉及循环的第一次或最后一次迭代。如果循环不像预期的那样工作，就检查循环是否多迭代或者少迭代了一次。循环多迭代或者少迭代一次，就说它存在**相差 1 错误**；这种错误是最常见的循环错误之一。注意，不要混淆<和<=。必须正确初始化循环。记住，循环有时需要迭代 0 次，所以要确定自己循环能正确处理这种情况。

无限循环常常是因为控制循环停止的布尔表达式错误而造成的。请检查是否弄反了不等式，混淆了小于和大于。无限循环的另一个常见根源是测试相等性来终止循环，而不是进行大于或小于比较。对于 double 类型的值，测试相等性不能提供有意义的答案，因为比较的只是一个近似值。即便是 int 类型，依赖相等性测试来终止循环也是非常危险的，因为只有一个数满足条件。

如果再三检查循环，确定没有错误，但程序依然不正确，就需要进行一些较复杂的测试。首先确定真的是循环的错误。程序错误并不一定能立即表现出来，一个地方的错误可能在另一个地方体现。将程序分解成不同的函数，就很容易确定错误的大概位置。

如果确定错误在特定循环中，应该在程序运行期间观察该循环如何更改变量值。这样

就知道循环正在做什么，并确定具体是什么错误。在程序运行期间观察变量值的变化，这称为对变量进行**跟踪**。许多系统都提供了调试工具，允许在不对程序进行任何修改的前提下跟踪变量。如果系统提供了这样的调试工具，就值得学习它的用法。如果系统没有提供，为了对变量进行跟踪，可在循环主体放一个临时的 cout 语句，每次循环迭代都将变量值写到屏幕上。

例定以下代码段需要调试：

```
int next = 2, product = 1;
while (next < 5)
{
    next++;
    product = product * next;
}
// 变量 product 包含从 2 到 5 的乘积
```

循环末尾的注释指出这个循环本来的用途，但通过测试，发现它为变量 product 赋了不正确的值。需要知道错在哪里。调试该循环需要跟踪变量 next 和 product。如果有调试工具，可以用工具简化调试。如果没有，可插入一个 cout 语句来跟踪变量，如下所示：

```
int next = 2, product = 1;
while (next < 5)
{
    next++;
    product = product * next;
    cout << "next = " << next
         << " product = " << product << endl;
}
```

跟踪变量 product 和 next，发现在第一次循环迭代之后，product 和 next 的值都是 3。显然，是从 3 而不是 2 开始计算乘积。

至少有两个很好的办法修正这个 bug。最简单的就是将变量 next 初始化为 1，而不是 2。这样一来，next 在循环中首次递增时，就会获得值 2，而不是 3。另一个办法是将递增语句放到乘法运算之后，如下所示：

```
int next = 2, product = 1;
while (next < 5)
{
    product = product * next;
    next++;
}
```

假定通过移动 next++的位置修正 bug。在添加了这个修正之后，工作并未完成。还必须再次测试修正过的代码。通过测试，发现结果依然不正确。重新跟踪变量，发现循环在乘以 4 之后就终止了，根本没有乘以 5。这样一来，就知道布尔表达式应该使用一个操作符<=，而不是操作符<。所以，正确代码如下：

```
int next = 2, product = 1;
while (next <= 5)
{
   product = product * next;
   next++;
}
```

每次修改程序都要重新测试。绝对不要假定一次修改就会使程序正确。发现一样东西

正确，并不表示其他所有东西都不需要纠正了。另外，正如这个例子所展示的，对程序的一个地方进行修改可能产生连锁反应，造成其他地方也需要修改。

> **测 试 循 环**
>
> 应该为每个循环提供不同的输入，从而测试以下循环行为：循环主体迭代 0 次；循环主体迭代 1 次；循环主体迭代最多的次数；循环主体迭代最多的次数减 1。这些只是最起码的测试。针对要测试的循环，也许还要进行其他特殊测试。

我们介绍的技术有助于在一个具有良好设计的程序中减少 bug 数量。然而，进行再多的调试，也无法将设计欠佳的程序转换成可靠而易读的程序。如果程序或算法很难理解，或者执行效率欠佳，就不要尝试修正它，而应将其抛弃，重新设计一个。你的目的是写一个容易阅读而且不大可能包含隐藏错误的程序。彻底抛弃设计有问题的代码，并从头再来，相较于对旧代码进行修正，往往能更快地生成一个能实际工作的、具有良好设计性能的程序。很多人都不明白这一点。虽然从短期来说，舍弃以前辛苦编写的代码有点儿可惜，但从长期来说，这样做利大于弊。事实上，被舍弃的代码并没有被白白浪费。通过编写它而获得的经验与教训将帮助你设计出更好的程序。相较于在没有任何经验的情况下编写程序，你现在往往能更快地写出一个具有良好设计性能的程序。糟糕的代码本身没有任何用处。

> **调试非常差的程序**
>
> 一团糟的程序就不要枉费心机调试了，直接扔了它，重新写。

√ 自测题

39. 跟踪变量是什么意思？怎样跟踪一个变量？
40. 什么是相差 1 循环错误？
41. 假定一段篱笆的长度是 100 英尺。每隔 10 英尺就立一个篱笆桩。总共需要立多少个篱笆桩？在一本讲编程的书中，这个问题为什么并不像表面上那么无聊？这个问题对于程序员有什么教育意义？

小　　结

- 布尔表达式的求值方式与算术表达式相似。
- 大多数现代编译器都支持值为 true 或 false 的 bool 类型。
- 可以写一个函数来返回 true 或 false。在 if-else 语句或者允许使用布尔表达式的任何地方，都可以将对该函数的调用作为一个布尔表达式来使用。
- 为了解决一个任务或者子任务，一个办法是写出条件，以及在每个条件之下需要采取的相应行动。在 C++中，可以用多路 if-else 语句来实现它。
- switch 语句是在程序实现一个菜单的好办法。
- 块(或代码块)是包含变量声明的一个复合语句。块中声明的变量是这个块的局部变量。在多路分支语句(比如一个多路 if-else 语句)的一个分支中，可以将块作为一个行动来使用。
- for 循环可实现“重复循环主体 n 次”。
- 通常用 4 种方法终止输入循环：已知长度的列表、每次迭代都询问、以哨兵值结束的列表和判断是否用完所有输入。
- 一般最好用伪代码设计循环，不要事先指定具体的 C++循环语句。一旦设计好算法，就很容易决定具体应该选择什么 C++循环语句。
- 为了简化嵌套循环，一个办法是将循环主体转变成函数调用。
- 一定要确保循环使用的变量在循环开始之前正确初始化。
- 一定要确保循环不会多迭代或者少迭代一次。
- 设计循环时，可考虑在循环主体中对关键变量进行跟踪。
- 如果程序或算法很难理解，或者执行效率很差，不要试图去修正它。彻底丢弃并重新设计一个。

自测题答案

1. a. true。
   b. true。注意，表达式(a)和(b)的含义是相同的。由于操作符==和<的优先级都要高于&&，所以不需要包括圆括号。但圆括号确实有助于提高代码的可读性。大多数人都认为相较于(b)的表达式，(a)的表达式更容易易懂，虽然两者作用相同。
   c. true。
   d. true。
   e. false。由于第一个子表达式(count == 1)的值是 false，马上就知道整个表达式的值是 false，不必对第二个子表达式求值。所以，x 和 y 的值无关紧要。这是短路求值，是 C++语言支持的。
   f. true。由于第一个子表达式(count < 10)的值是 true，马上就知道整个表达式的值是 true，不必对第二个子表达式求值。所以，x 和 y 的值无关紧要。这称为短路求值，是 C++语言支持的。
   g. false。注意(g)的表达式将(f)的表达式作为自己的一个子表达式。子表达式根据短路求值的原理进行求值，所以(g)的整个表达式等价于：

```
!( (true || (x < y)) && true )
```

   进而等价于!(true && true)，进而等价于!(true)，终值为 false。
   h. 这个表达式在求值时会产生错误，因为第一个子表达式((limit/count) > 7)会造成除以 0 错误。
   i. true。因为第一个子表达式(limit < 20)的值是 true，马上就知道整个表达式的值是 true，不必对第二个子表达式求值。因此，第二个子表达式根本不会求值，计算机会忽略它的除以 0 错误：

```
((limit/count) > 7)
```

   这称为短路求值，是 C++语言支持的。
   j. 这个表达式在求值时会产生错误，因为第一个子表达式((limit/count) > 7)会造成除以 0。
   k. false。因为第一个子表达式(limit < 0)的值是 false，马上就知道整个表达式的值是 false，不必对第二个子表达式求值。因此，第二个子表达式根本不会求值，计算机会忽略它的除以 0 错误：

```
((limit/count) > 7)
```

   这称为短路求值，是 C++语言支持的。
   l. 这个表达式是无意义的。表达式没有直观的意义，但 C++会将 int 值转换成布尔值，再执行&&和!运算。所以，虽然没有意义，但 C++仍会对它进行求值。记住，在 C++中，任何非零的整数都转换成 true，0 转换成 false。针对以下表达式：

```
(5 && 7) + (!6)
```

   C++会按照以下顺序来求值：在表达式(5 && 7)中，5 和 7 被转换成 true，true && true 的结果是 true，C++再把它转换成 1。在(!6)中，6 被转换成 true，所以 !(true)的结果是 false，C++会把它转换成 0。因此，整个表达式相当于 1 + 0，结果就是 1。C++可以将数字 1 再次转换成 true，但如果说答案是 true，可能让人产生迷惑。所以，说答案是 1 可能会更好一些。

   没必要在这些无意义的表达式上钻牛角尖儿，但偶尔体验一下，有助于理解假如在一个表达式中不正确地混合数值和布尔操作符，编译器为什么不显示错误消息。

2. 到目前为止，我们已经学习了分支语句、循环语句和函数调用语句。
   学过的分支语句的例子是 if 和 if-else 语句。循环语句的例子是 while 和 do-while 语句。

3. 2 < x < 3 是合法的 C++表达式，但它并不是像许多人想象的那样表示(2 < x) && (x < 3)，而是表示(2 < x) < 3。由于(2 < x)是布尔表达式，所以值要么为 true，要么为 false，最后会转换成 1 或 0。因此，不管 x 的值是什么，2 < x < 3 肯定为 true(1 和 0 都小于 3)。换言之，无论 x 等于多少，最终输出的都是“true”。

4. 不会。布尔表达式 j > 0 为 false(j 值为-1)。&&会使用短路求值，所以只要第一个表达式的求值结果是 false，就不再需要对第二个表达式进行求值，终值肯定为 false。由于第二个表达式不会求值，所以不会产生除以 0 错误。

5.
```
Start
Hello from the second if.
End
Start again
End again
```

6.
```
large
```

7.
```
small
```

8.
```
medium
```

9.
```
Start
Second Output.
End
```

10. 不管布尔表达式是 (x > 10) 还是 (x > 100)，结果都是一样的。输出与自测题 9 完全相同。

11.
```
Start
100
End
```

12. 以下两种方案都是正确的。

方案 1

```
if (n < 0)
    cout << n << " is less than zero.\n";
else if ((0 <= n) && ( n <= 100))
    cout << n << " is between 0 and 100 (inclusive).\n";
else if (n > 100)
    cout << n << " is larger than 100.\n";
```

方案 2

```
if (n < 0)
    cout << n << " is less than zero.\n";
else if (n <= 100)
    cout << n << " is between 0 and 100 (inclusive).\n";
else
     cout << n << " is larger than 100.\n";
}
```

13. 除非特别指定，否则 enum 常量的默认值从 0 开始，并依次递增 1。输出是 3 2 1 0。

14. 部分 enum 常量值已经指定。没有指定的常量在上一个值的基础上增 1，所以输出是 2 1 7 5。

15.
```
Roast worms
```

16.
```
Onion ice cream
```

17.
```
Chocolate ice cream
Onion ice cream
```

(这是因为 case3 中没有 break 语句。)

18.
```
Bon appetit!
```

19.
```
42 22
```

20. 可以稍微改变一下代码，帮助自己理解在每次输出时，具体使用的是 x 的哪一个声明。

```
{
    int x1 = 1; // 在这一列输出
    cout << x1 << endl; // 1<cr>
    {
        cout << x1 << endl; // 1<cr>
        int x2 = 2;
        cout << x2 << endl; // 2<cr>
        {
```

```
        cout << x2 << endl; // 2<cr>
        int x3 = 3;
        cout << x3 << endl; // 3<cr>
    }
    cout << x2 << endl; // 2<cr>
}
cout << x1 << endl; // 1<cr>
```

这里的<cr>指的是输出的换行处理。

21.
```
2 1 0
```

22.
```
2 1
```

23.
```
1 2 3 4
```

24.
```
1 2 3
```

25.
```
2 4 6 8
```

26.
```
Hello 10
Hello 8
Hello 6
Hello 4
Hello 2
```

27.
```
2.000000 1.500000 1.000000 0.500000
```

28. a. 使用 for 循环。

    b. 和 c. 两者都需要 while 循环，因为输入列表可能是空的。

    d. 需要一个 do-while 循环，因为至少需要执行一次测试。

29. a.
```
for (int i = 1; i <= 10; i++)
if (i < 5 && i != 2)
    cout << 'X';
```

    b.
```
for (i = 1; i <= 10; i = i + 3)
    cout << 'X';
```

    c.
```
cout << 'X'; // 这是为了保持输出结果相同。还要注意对 m 的初始值的修改
for (long m = 200; m < 1000; m = m + 100)
    cout << 'X';
```

30. 输出是 1024 10。10 是以 2 为底 1024 的对数。

31. 输出是 1024 1。for 之后的分号(;)可能是一个人为的疏忽。

32. 这是一个无限循环。重点是变量更新表达式 i = i * 2。如果 i 的初始值是 0，那么 i 值永远都是 0。

33. 4 3 End of Loop

34. 4 3

    注意，由于 exit 语句终止整个程序，所以不会输出"End of Loop."。

35. break 语句用于终止循环(while，do-while 或者 for 语句)，或者终止 switch 语句中的一个 case。除此之外，break 在 C++程序的其他任何地方都不合法。注意，如果循环是嵌套的，break 就只终止当前那一级循环。

36.
```
for (int count = 1; count <= 10; count++)
    cout << "Hello\n";
```

37. 可用任何奇数作为哨兵值。

```
int sum = 0, next;
cout << "Enter a list of even numbers. Place an\n"
     << "odd number at the end of the list.\n";
cin >> next;
while ((next % 2) == 0)
{
   sum = sum + next;
   cin >> next;
}
cout << "Sum = " << sum << endl;
```

38. 输出太长，这里无法全部列出。但模式基本如下：

```
1 times 10 = 10
1 times 9 = 9
   ...
1 times 1 = 1
2 times 10 = 20
2 times 9 = 18
   ...
2 times 1 = 2
3 times 10 = 30
   ...
```

39. 跟踪一个变量是指在程序运行期间，观察变量值的变化情况。可用特殊调试工具进行跟踪，或者在程序中插入临时性的输出语句进行跟踪。

40. 如果循环主体多迭代或者少迭代一次，就说发生了相差 1 错误。

41. 解决任何问题时都有可能出现相差 1 错误，这种错误并非循环特有。一些人会轻率地得出以下推论：

10 根篱笆桩 = 100 英尺的篱笆 / 每根桩子相距 10 英尺

但这个结果是错误的，因为按照这个答案，最后 10 英尺的篱笆将没有桩子。事实上，要在 100 英尺的篱笆中，每 10 英尺就立一根桩，总共需要 11 根篱笆桩。

编 程 练 习

编程练习一般只需写很小的程序，运用本章提到的编程概念。

1. 写程序模拟“石头－剪子－布”游戏。两个玩家分别输入 R(代表石头，Rock)，S(代表剪子，Scissor) 或者 P(代表布；英语国家不用布而用纸，即 Paper，这里根据中国习惯进行了转换)，然后由程序判断谁赢了。基本规则是：布包石头，石头砸坏剪子，而剪子剪布。如果选择一样，就算平局。要允许用户使用小写或大写字母输入 R，S 或者 P。你的程序应该包括一个循环，每次都让用户选择是否想重新玩一遍。

2. 写程序计算循环信贷账户的到期利息、总应付款以及最小付款额。程序接收账户余额作为输入，它加上利息就等于总应付款。利率表是这样的：1～1000 美元的利率为 1.5%，1000 美元以上的部分为 1%。如果总应付款是 10 美元或者 10 美元以下，那么最小付款就等于总应付款；否则，就是 10 美元，或者总欠款的 10%，取其中较多的。你的程序应该包括一个循环，允许用户重复这个计算，直到他们表示操作完毕。

3. 写一个算命程序。用户输入生日，程序报告相应的星座和运势。要了解运势和星座日期，请查询网络或报纸[①]。然后对程序进行改进，如果生日相距一个星座非常近，只有一两天，程序应该宣布这个生日位于两个星座的“交点”，并针对相距较近的那个星座也输出相应的占卜内容。这个程序可能包含一个较长的多路分支。程序应包括一个循环，允许用户重复计算，直到他们表示操作完毕。

   下面总结了星座与日期的对应关系：

① 推荐访问 http://www.d1xz.net/了解相关知识。——译注。

Aries(白羊座/牡羊座)	3 月 21 日～4 月 19 日
Taurus(金牛座)	4 月 20 日～5 月 20 日
Gemini(双子座)	5 月 21 日～6 月 21 日
Cancer(巨蟹座)	6 月 22 日～7 月 22 日
Leo(狮子座)	7 月 23 日～8 月 22 日
Virgo(处女座)	8 月 23 日～9 月 22 日
Libra(天秤座)	9 月 23 日～10 月 22 日
Scorpio(天蝎座)	10 月 23 日～11 月 21 日
Sagittarius(射手座)	11 月 22 日～12 月 21 日
Capricorn(魔羯座)	12 月 22 日～1 月 19 日
Aquarius(水瓶座)	1 月 20 日～2 月 18 日
Pieces(双鱼座)	2 月 19 日～3 月 20 日

4. 同一种元素的星座最合得来。星座学描述了 4 种元素，每种元素都包含 3 个星座。这些元素是“火”(白羊座、狮子座、射手座)、“土”(金牛座、处女座、魔羯座)、“气”(双子座、天秤座、水瓶座)和“水”(巨蟹座、天蝎座、双鱼座)。

   根据占星理论，和你最合得来的是同一星座的人，或者同一元素的另两个星座。例如，金牛座的人最合得来的是其他金牛座的人，以及另外两个“土”系星座的人，即处女座和魔羯座。

   修改编程项目 3 的程序，使之显示与输入的生日最合得来的星座的名称。

5. 写程序找出并打印 3 到 100 的所有质数。质数是只能被 1 和它自己整除的数，比如 3，5，7，11，13，17，……

   解决该问题的一个办法是使用一个双重嵌套循环。外层循环遍历从 3 到 100 的数，内层循环判断外层循环的当前计数器值是不是质数。为了判断 n 是不是质数，一个办法是从 2 循环到 $n-1$，如果 n 能被其中的任何一个数整除，表明 n 不是质数。如果 n 不能被从 2 到 $n-1$ 的任何一个数整除，那么 n 肯定是质数(注意，有几个简单的技巧可以提高这个算法的效率)。

6. 阿基米德定理指出，一个物体浸泡在液体中，其浮力等于被它排开的液体的重量。浮力计算公式如下：

$$F_b = V \times y$$

   其中，F_b 是浮力，V 是浸泡在液体中的物体的体积，y 是液体的比重。如果 F_b 大于或等于物体的重量，它就会浮起来，否则会沉没。

   写程序来输入一个球体的重量(单位：磅)和半径(单位：英尺)，输出该球体在水中是否会沉没。水的比重 $y = 62.4$ 磅/英尺^3。球体体积公式是 $V=(4/3)\pi r^3$。

7. 写程序找出在华氏温度和摄氏温度中数值相同的一个温度。摄氏温度(C)转换成华氏温度(F)的公式是：

$$F = \frac{(9 \times C)}{5} + 32$$

   程序应创建两个整数变量来表示摄氏温度和华氏温度。将温度初始化为 100 摄氏度。在一个循环中，使摄氏温度递减，并计算对应的华氏温度，直至两个值相等。[①]

   由于处理的是整数值，所以公式可能无法为每一个可能的摄氏温度都给出确切的结果。但就本题来说，这个问题影响不了你的解决方案。

编 程 项 目

编程项目要求综合运用多方面的知识来解决问题，程序一般比编程练习大，解题方式多样化。

1. 写程序计算长途电话费。电话费根据以下费率表来确定：
   a. 周一到周五，8:00 A.M.到 6:00 P.M.之间打出的任何电话都按每分钟 0.40 美元收取；

① 本题没有说一定是零上温度。——译注

b. 周一到周五，8:00 A.M.之前或者 6:00 P.M.之后打出的任何电话都按每分钟 0.25 美元收取；
c. 周六或周日打出的任何电话都按每分钟 0.15 美元收取；

在输入中，应该指定当前是星期几、开始通话时间，以及以分钟为单位的通话时间。输出电话费金额。时间以 24 小时格式输入，假定时间是 1:30 P.M.，输入如下：

```
13:30
```

输入星期几采用以下格式，把它们存储到 char 类型的一个变量中：

```
Mo Tu We Th Fr Sa Su
```

务必允许用户采用大写、小写或大小写混合形式输入。通话时间用 int 类型的变量输入(假定用户已将时间取整为一个整数分钟数)。程序应包括一个循环，允许用户重复这一计算，直到他们表示操作完毕。

2. (本题要求掌握复数的基础知识；所以，只有在数学课上学过复数，才适合做这道题)
写 C++程序解二次方程，求它的根。对于一个二次方程来说(a 不等于 0)：

$$ax^2 + bx + c = 0$$

它的根是：

$$(-b \pm \text{sqrt}(b^2 - 4ac)) / 2a$$

判别式($b^2 - 4ac$)的值决定了根的本质。判别式的值等于 0，方程有两个相等的实根；判别式的值为正，方程有两个不等的实根；判别式的值为负，方程没有实根，但有两个复数根。

程序获取 a，b 和 c 的值并输出根。输出复数根要多动一下脑子。使用循环，允许用户重复计算，直到表示操作完毕。

3. 写程序接收 4 位阿拉伯数字年份，输出罗马数字年份。一些重要的罗马数字包括：V 代表 5，X 代表 10，L 代表 50，C 代表 100，D 代表 500，M 代表 1000(I，II 和 III 分别代表什么就不用说了吧)。记住某些数字是通过“减去”一个罗马“数字”而生成的；例如，IV 代表 4，通过 V 减 I 生成，XL 代表 40，CM 代表 900，依此类推。一些示范性的年份转换结果是：MCM 代表 1900，MCML 代表 1950，MCMLX 代表 1960，MCMXL 代表 1940，MCMLXXXIX 代表 1989。假定年份在 1000～3000 之间。你的程序应该包括一个循环，允许用户重复计算，直到表示操作完毕。

4. 写程序统计一手黑杰克(21 点)的点数。玩家可以拿 2～5 张牌(玩家自己决定多少张牌，但这对本题来说没有影响)。牌点为 2～10 的每张牌都分别计为相应的点数。花牌(J，K 和 Q)计为 10 点。目标是尽可能接近但不要超过 21 点。任何超过 21 点的点数都表示这一手牌已经“爆”了。A 既可以计为 1 点，也可计为 11 点，选择对玩家最有利的。例如，1 张 A 和 1 张 10 可计为 11 点或者 21 点，但由于 21 点更好，所以这一手牌要计为 21 点。1 张 A 和 2 张 8 可计为 17 点或者 27 点。由于 27 点已经“爆”了，所以这一手牌要计为 17 点。

程序首先询问用户手上有几张牌，用户输入 2，3，4 或 5。然后询问每张牌的点数。点数可为 2～10，J，K，K 和 A。对输入进行处理的一种较好的方式是使用 char 类型。所以 2 被作为字符'2'读入，而不是数字 2。2～9 的值分别作为字符'2'～'9'输入。10、J、K 和 A 等值分别作为字符't'，'j'，'q'，'k'和'a'输入(当然，用户不需要输入单引号)。一定要同时支持大写和小写。

读取这些值后，程序将它们从字符值转换为牌点数值，注意 A 需要特殊处理。输出的可能是 2～21 的一个数字，也可能是“爆了”的显示。可能需要一个或者多个 switch 或嵌套 if-else 多路分支语句。程序应包括一个循环，允许用户重复计算，直到表示操作完毕。

5. 分期贷款利息是根据不断减少的余额来支付的，所以假定一笔贷款的利率为 14%，那么在经过多次分期还款之后，总共支付的利息金额肯定小于当初贷款额的 14%。写一个程序，获取贷款额和利率作为输入，输出每个月的还款额以及贷款余额，直到偿还所有贷款为止。假定每个月偿还原始贷款额的 1/20；在其中减去利息之后，剩下的就是每个月偿还的本金，需要在贷款余额中减去这个本金。因此，假定贷款额是 20000 美元，每个月固定还款 1000 美元。假定利率是 10%，那么每个月的利息就是贷款余额的 10%的 1/12。第一个月，(20000 的 10%)/12 即 166.67 美元用于支付利息，剩余的 833.33 美元就是偿还的本金，所以第一个月还款之后，贷款余额变成(20000 － 833.33) = 19166.67 美元。下个月的利息是(19166.67 的 10%)/12，依此类推。另外，让程序输出贷款期间支付的利息总额。

最后，计算每年支付的利息金额占原始贷款额的百分比。例如，假定为 10000 美元的一笔贷款支付了

1000 美元的利息，总共两年还清，那么每年利息是 500 美元，也就是 10000 美元贷款额的 5%。程序允许用户根据需要重复任意次数。

6. 斐波那契数字 F_n 是这样定义的：F_0 等于 1，F_1 等于 1，而

$$F_{i+2} = F_i + F_{i+1}$$

其中，$i = 0$，1，2，…。换言之，每个数都是前两个数之和。前几个斐波那契数字是 1，1，2，3，5 和 8。出现斐波那契数字的一个场合是特定的生物种群生长率。假定一个种群没有死亡，这个数列就显示了在经历每个周期之后，这个种群会发展到什么规模。生物体花费两个时间段来繁殖下一代，以后每个周期都会繁殖另一代。例如无性繁殖，它的繁殖速度就是每个周期都产生新的一代。

假定绿锈以这个程度蔓延，每个周期为 5 天。所以，如果绿锈最开始有 10 磅，那么 5 天之后仍然有 10 磅；10 天之后变成 20 磅；15 天之后变成 30 磅；20 天变成 50 磅，依此类推。写程序要求用户输入绿锈的初始磅数和一个天数。输出在经过指定天数后，绿锈的总磅数。忽略绿锈平滑蔓延的过程，假定它们在 4 天的时间里都保持不变，但每到第 5 天就增加。程序允许重复计算任意次数。

7. e^x 的值可通过以下公式大致计算出来：

$$1 + x + x^2/2! + x^3/3! + \cdots + x^n/n!$$

写程序获取 x 的值作为输入，输出 n 值从 1 变化到 100 时，上述公式每一次的求值结果。程序还应使用预定义函数 `exp` 输出 e^x 的值。`exp(x)` 返回的就是 e^x 的约值。`exp` 函数在头文件为 `cmath` 的库中。程序允许用户选择新的 x 值，并重复计算，直到用户表示操作完毕。

使用 `double` 类型的变量存储阶乘，否则可能产生整数溢出(或者调整你的计算，避免直接计算阶乘)。调整输出，不要输出 100 行，而是输出 10 行，每行 10 个值。

8. pi 的值可通过以下公式大致计算出来：

$$pi = 4 [1 - 1/3 + 1/5 - 1/7 + 1/9\cdots + ((-1)^n) / (2n + 1)]$$

写 C++程序使用以上数列计算 pi 的近似值。程序要获取输入值 n。程序应包含一个循环，允许用户输入新的 n 值来重复计算，直到要求终止程序。

9. 视频讲解：*Solution to Programming Project 3.9*

以下问题有时称为“蒙特霍问题”或“三门问题”(The Monty Hall Problem)。你是一个电视游戏节目的参加者，并有机会赢得大奖。你面前有三扇关着的门。一扇门后是一辆崭新的汽车。另外两扇门后则是一些安慰奖。奖品位置随机选择。节目主持人要求你选一扇门，所以你选了一扇。但在揭示门后有什么东西之前，主持人先打开了另外两扇门之一，露出门背后的安慰奖。这个时候，主持人会问你是想保持原先的选择，还是改为选择另一扇关的门。如何选择才能优化赢得汽车的概率？保持初始选择或者换一扇门，会影响赢得大奖的概率吗？

写一个模拟程序来解决这个问题。程序应该模拟这个问题 10000 次，每次都随机选择奖品位置，并统计在保持初始选择的情况下赢得汽车的次数，以及在换门的情况下赢得汽车的次数。输出两种策略下的获奖概率。程序必须准确地模拟选择门，打开门，然后改变门的过程。不要对实际的解决方案做出任何假定(例如，假定有 1/3 或者 1/2 的概率赢得奖品)。

生成随机数的库函数参见附录 4。更详细的描述参见第 4 章。

10. 修改第 2 章的编程项目 13，询问用户每天的运动情况：

a. Sedentary – 好静
b. Somewhat active (exercise occasionally) – 较活跃(偶尔锻炼)
c. Active (exercise 3–4 days per week) – 活跃(每周锻炼三、四天)
d. Highly active (exercise every day) – 非常活跃(每天都锻炼)

用户回答“Sedentary”，将计算的 BMR 增大 20%。回答“Somewhat active”，增大 30%。回答“Active”，增大 40%。回答“Highly active”，增大 50%。根据新的 BMR 值输出巧克力条的数量。

11. 视频讲解：*Solution to Programming Project 3.11*

烤箱上的按键用于输入烘焙温度，布局和电话按键相似：

1	2	3
4	5	6
7	8	9
	0	

遗憾的是，由于电路损坏，最左边那一列的数字无法工作，即 1，4 和 7 键坏了。如果食谱要求无法输入的温度，可考虑替换成一个能输入的。写程序要求用户输入希望的温度。温度必须在 0 到 999 度之间。如果希望的温度不包含 1，4 或 7，就直接输出这个温度。否则，计算并输出第一个比它大和比它小而且不包含 1，4 和 7 的温度。例如，如果希望的温度是 450，程序应输出 399 和 500。类似地，如果希望的温度是 375，应输出 380 和 369。

12. “23”游戏是一个双人游戏，道具是 23 根牙签。玩家轮流取 1，2 或 3 根牙签。拿到最后一根牙签的是输家。写程序和计算机玩“23”。总是玩家先走。轮到计算机时，它根据以下规则采取行动：

    - 如果剩余牙签多于 4 根，计算机就取走 4 – X 根，X 是玩家上次取走的牙签数。
    - 如果剩余 2~4 根牙签，计算机取走足够多的牙签，确保只剩下 1 根。
    - 如果剩余 1 根牙签，计算机只能取走它并认输。

    玩家输入要取走的牙签数量时，程序应对输入的有效性进行检查。要确定玩家输入的数在 1 到 3 之间，而且试图取走的不能超过当前剩余的。

13. 蝙蝠侠的老对头谜语人正计划对华府宾夕法尼亚大道发动下一次袭击。按照惯例，他将地址留在了一则谜语中。宾夕法尼亚大道的门牌号是一个 4 位数，其中：

    - 4 个数位都不同
    - 千位数是十位数 3 倍
    - 整个数字是奇数
    - 所有数位之和是 27

    写程序用循环(可能要多个循环)找出谜语人计划袭击的地址。

14. 一个增强现实(AR)游戏要求抓住 Edoc 并获得 Edoc 糖。12 块糖将一只 Edoc 进化成 Margorp。每次进化返 1 块糖和 500 经验值。一个 Edoc 或 Margorp 能转化成 1 块糖。为方便游戏玩家，写一个 Edoc 计算器程序，输入抓住的 Edoc 的数量和持有的 Edoc 糖的数量，假定 Margorp 初始数量为 0。程序最终输出能通过转化和进化获得的最大经验值。Edoc 进化成 Margorp 后，程序应考虑是否转化 Margorp 来获得足够多的糖使更多 Edoc 进化(目的是获取经验值)。例如，假定现有 71 块糖和 53 个 Edoc，程序输出如下所示。注意可按其他次序转化和进化，造成 Edoc 和 Margorp 最终数量不定，但能获得的最大经验值是一样的。

```
转化 37 Edoc 和 0 Margorp，现有：
   108 糖，16 Edoc 和 0 Margorp
进化 9 Edoc 获得 4500 经验值，现有：
   9 糖，7 Edoc 和 9 Margorp
转化 0 Edoc 和 9 Margorp，现有：
   18 糖，7 Edoc 和 0 Margorp
进化 1 Edoc 获得 500 经验值，现有：
   7 糖，6 Edoc 和 1 Margorp
转化 4 Edoc 和 1 Margorp，现有：
   12 糖，2 Edoc 和 0 Margorp
进化 1 Edoc 获得 500 经验值，现有：
   1 糖，1 Edoc 和 1 Margorp
总经验值 = 5500
```

第 4 章　过程抽象和返回值的函数

还有一位最巧妙的建筑师，他发明了一种建造房屋的新方法，即先从屋顶造起，自上而下一路盖到地基。

——约拿旦•斯威夫特，《格列佛游记》

概述

程序可以看作由若干个小的子部分组成。在这些子部分中，有的获取输入数据，有的计算输出数据，有的则显示输出数据。和大多数编程语言一样，C++提供了单独命名和编写这些子部分的工具。在 C++中，这些子部分称为**函数**。C++有两种主要的函数，本章介绍其中一种函数(也就是用于计算单独一个值的函数)的基本语法，同时讨论这些函数如何为程序设计提供帮助。下面首先讨论基本的设计原则。

预备知识

阅读本章之前，应该完成第 2 章的学习，而且至少应该浏览一下第 1 章。

4.1　自顶向下设计

记住，为了写一个程序，首先要构思出程序的用途，再用通俗易懂的文字写下这种方法。这个过程类似于拟定要由办事员遵照办理的指令。如第 1 章所述，这组指令称为**算法**。设计算法最有效的手段就是将任务分解成多个子任务，再将每个子任务分解成更小的子任务，依此类推。最终，子任务会变得很小，很容易用 C++代码实现。这种设计方法称为**自顶向下设计**。有时也称为**逐步求精**，或者更形象地称为**分而治之**。

使用自顶向下方法，可将程序的任务分解为一组子任务，并通过子算法来解决这些子任务，从而完成一个程序的设计。在 C++程序里采用这种自顶向下结构，能使程序更容易理解，更容易更改，而且使程序更清晰，更容易编写、测试和调试。和大多数编程语言一样，C++提供了在程序中包括独立的子部分的机制。在其他编程语言里，这些子部分称为子程序、子过程或者方法。在 C++里，这些子部分称为**函数**。

通过函数将编程任务分解为子任务的好处在于，不同的人可以负责不同的子任务。构造大型程序(比如编译器或办公管理系统)时，如果程序开发时间有限，这种形式的团队协作就非常重要。本章首先介绍如何使用别人写好的函数。

4.2　预定义函数

C++提供了预定义函数库，可在自己的程序中使用这些现成的函数。在解释如何定义函数之前，先来看看如何使用已定义好的函数。

使用预定义函数

下面以 sqrt 函数为例解释如何使用预定义函数。sqrt 函数获取一个值(如 9.0)，并计算它的平方根(如 3.0)。函数获取的值称为函数的**实参**[①]，它计算后得到的值称为**返回值**。有的函数可能有多个实参，但函数返回值不能超过 1 个(可以是 0 个)。将函数视为小程序，实参就相当于输入，返回值相当于输出。

在程序中使用函数时采用的语法很简单。例如，要把 theRoot 变量设为 9.0 的平方根，可使用以下赋值语句：

```
theRoot = sqrt(9.0);
```

表达式 sqrt(9.0) 称为**函数调用**。函数调用中的实参可以是常量，如 9.0，也可以是变量，甚至可以是复杂的表达式。函数调用本身是表达式，可像其他任何表达式那样使用。但凡需要提供一个值的地方都可以进行函数调用，只要函数返回值具有正确的类型。例如，sqrt 的返回值是 double 类型，所以以下语句完全合法(就是有点小气[②])：

```
bonus = sqrt(sales) / 10;
```

sales (销售额) 和 bonus (奖金) 通常是 double 类型的变量。函数调用 sqrt(sales) 被视为单独一项，好比用圆括号把它封闭起来一样。所以，上述赋值表达式等价于：

```
bonus = (sqrt(sales))/10;
```

还可在 cout 语句中直接使用函数调用，如下所示：

```
cout << "The side of a square with area " << area
     << " is " << sqrt(area);
```

图 4.1 是使用了预定义函数 sqrt 的一个完整程序。程序根据用户的预算计算最多能购买多大的一个正方形狗舍。程序要求输入预算金额，根据金额计算最大能购买占地多少平方英尺的正方形狗舍。然后，根据正方形的面积，用 sqrt 函数计算出边长。

图 4.1　函数调用

```
1  //根据用户的预算，计算可以购买多大的狗舍
2
3  #include <iostream>
4  #include <cmath>
5  using namespace std;
6
7  int main()
8  {
9      const double COST_PER_SQ_FT = 10.50; // 狗舍每平方英尺价格
10     double budget, area, lengthSide;
11
12     cout << "Enter the amount budgeted for your dog house $";
13     cin >> budget;
14
15     area = budget/COST_PER_SQ_FT;
16     lengthSide = sqrt(area);
17
```

① 在必要的时候，本书对实参和形参进行了区分。为帮助你理解，这里简单解释一下。实参是实际向函数传递的数据，比如值、表达式或者变量等。相反，形参只是定义函数时指定的占位符，调用函数时要用实参代替它。后文讲到“程序员自定义函数”时，会进一步解释形参的问题。——译注

② 因为奖金提成只有销售额的一成。——译注

```
18      cout.setf(ios::fixed);
19      cout.setf(ios::showpoint);
20      cout.precision(2);
21      cout << "For a price of $" << budget << endl
22           << "I can build you a luxurious square dog house\n"
23           << "that is " << lengthSide
24           << " feet on each side.\n";
25
26      return 0;
27  }
```

示范对话

```
Enter the amount budgeted for your dog house $25.00
For a price of $25.00
I can build you a luxurious square dog house
that is 1.54 feet on each side.
```

函 数 调 用

函数调用是表达式，由函数名和用圆括号括起来的实参构成。假如函数调用中的实参不止一个，就用逗号分隔这些实参。由于函数调用是表达式，所以可以像其他任何表达式那样使用。唯一要注意的是，在使用函数的那个位置，该函数的返回值的类型必须是合法的。

语法

Function_Name(*Argument_List*)

其中，*Argument_List* 是以逗号分隔的实参列表：

Argument_1, Argument_2, …, Argument_Last

示例

```
side = sqrt(area);
cout << "2.5 to the power 3.0 is "
     << pow(2.5, 3.0);
```

注意，图 4.1 出现了一个新元素，如下所示：

```
#include <cmath>
```

它看起来和以下语句非常相似：

```
#include <iostream>
```

事实上，上述两个语句的性质一样。如第 2 章所述，像这样的语句称为**预编译指令**。尖括号(< >)内的名称是**头文件**的名称。一个库的头文件为编译器提供了关于这个库的特定信息，include 预编译指令将这些信息传递给编译器。这样一来，链接器就能在库中找到函数的目标码，以便将库正确链接到你的程序。例如，iostream 库包含 cin 和 cout 的定义，库的头文件称为 iostream。math 库包含 sqrt 和其他许多数学函数的定义，库的头文件称为 cmath。要在程序中使用某个库的预定义函数，就必须用预编译指令包含那个库的头文件，如下所示：

```
#include <cmath>
```

请务必遵守示例所采用的语法。不要遗漏符号<和>。在<和文件名之间，文件名和>之间，不能有空格。此外，有的编译器要求符号#前后都不能有空格。所以，最保险的做法就

是将符号#放在一行的最开头，而且不要在符号#和 include 之间插入任何空格。这些#include 预编译指令通常集中在程序文件的起始位置。

以前还说过，以下预编译指令：

```
#include <iostream>
```

要求同时使用以下 using 预编译指令：

```
using namespace std;
```

这是因为在 iostream 库中定义 cin 和 cout 这样的名称时，规定这些名称是 std 命名空间的一部分。大多数标准库都做出了类似的规定。所以，一旦使用 include 预编译指令包含了标准库，比如：

```
#include <cmath>
```

就可能需要同时使用一个 using 预编译指令：

```
using namespace std;
```

如果有多个 include 预编译指令，则不必重复上述 using 指令。

通常，使用库唯一要做的就是在程序文件中添加 include 和 using 指令。如果 include 和 using 指令有效，就不必给出其他指令。但是，某些系统的某些库可能要求为编译器提供额外的指令，或者显式运行链接程序来链接一个库。早期的 C 和 C++编译器不会自动搜索所有要链接的库。具体情况因系统而异。所以，如有必要，请查阅手册，或咨询这方面的专家。

有人会告诉你，include 预编译指令不由编译器来处理，而是由**预处理器**处理。这种说法也对，但它更像是一种文字游戏，对你而言意义并不大。编译程序时，几乎所有编译器都会自动调用预处理器。

图 4.2 总结了几个预定义函数，更多的预定义函数请参见附录 4。

图 4.2　一些预定义函数

名称	说明	实参类型	返回值类型	示例	值	库的头文件
sqrt	平方根	double	double	sqrt(4.0)	2.0	cmath
pow	乘方	double	double	pow(2.0, 3.0)	8.0	cmath
abs	取 int 值的绝对值	int	int	abs(-7) abs(7)	7 7	cstdlib
labs	取 long 值的绝对值	long	long	labs(-70 000) labs(70 000)	70 000 70 000	cstdlib
fabs	取 double 值的绝对值	double	double	fabs(-7.5) fabs(7.5)	7.5 7.5	cmath
ceil	向上取整	double	double	ceil(3.2) ceil(3.9)	4.0 4.0	cmath
floor	向下取整	double	double	floor(3.2) floor(3.9)	3.0 3.0	cmath
srand	为随机数生成器提供种子值	无	无	srand()	无	cstdlib
rand	随机数	无	int	rand()	0~RAND_MAX	cstdlib

注意，绝对值函数 abs 和 labs 以及随机数函数 srand 和 rand 在头文件为 cstdlib 的库中。所以，任何程序要使用这些函数，必须包含以下预编译指令：

```
#include <cstdlib>
```

除此之外，表格中的其他所有函数都在头文件为 cmath 的库中，和 sqrt 函数一样。

注意有三个绝对值函数。生成 int 类型的用 abs，生成 long 类型的用 labs，生成 double 类型的则用 fabs。更麻烦的是，abs 和 labs 在头文件为 cstdlib 的库中，而 fabs 在头文件为 cmath 的库中。fabs 是 floating-point absolute value(浮点绝对值)的缩写。记住，带小数的数字(比如 double 类型的数字)通常称为**浮点数**。

预定义函数的另一个例子是 pow，它包含在头文件为 cmath 的库中。在 C++中，pow 函数用于求乘方。例如，以下语句将变量 result 设为 x^y：

```
result = pow(x, y);
```

因此，以下 3 行代码在屏幕上输出数字 9.0，因为 $(3.0)^{2.0}$ 等于 9.0：

```
double result, x = 3.0, y = 2.0;
result = pow(x, y);
cout << result;
```

注意，以上 pow 函数调用返回 9.0，而不是 9。pow 函数总是返回 double 而非 int 值。还要注意 pow 函数要求两个实参。函数可以有任意数量的实参。每个实参的类型都是固定的，函数调用时提供的实参应该和该类型匹配。许多时候，如果使用了不同类型的实参，C++将执行自动类型转换，但结果可能不是你希望的。调用函数时，应该使用为那个函数指定的实参类型，但一个例外是从 int 到 double 的自动类型转换。许多时候，在调用 pow 函数时，可以安全地使用 int 类型的实参，即使要求的是 double 类型。

pow 函数的许多实现对允许的实参进行了限制。有的实现要求假如第一个实参为负，则第二个实参必须是整数。学习编程时，由于还有其他许多东西需要考虑，所以最简单、最保险的做法就是：只有在第一个实参非负的时候才使用 pow 函数。

随机数生成

视频讲解：*Random Number Generation*

游戏和模拟程序经常需要生成随机数。C++提供了生成伪随机数的预定义函数。**伪随机数**是表面上随机，实则由可预见的公式决定的数。例如，下面是非常简单的一个伪随机数生成器的公式，它根据生成的上一个随机数 R_{i-1} 决定第 i 个随机数 R_i：

$$R_i = (R_{i-1} \times 7) \% 11$$

将初始“种子”值 R_0 设为 1。像下面这样获取第一个“随机”数 R_1：

$$R_1 = (R_0 \times 7) \% 11 = (1 \times 7) \% 11 = 7 \% 11 = 7$$

获取第二个“随机”数 R_2：

$$R_2 = (R_1 \times 7) \% 11 = (7 \times 7) \% 11 = 49 \% 11 = 5$$

获取第三个“随机”数 R_3：

$$R_3 = (R_2 \times 7) \% 11 = (5 \times 7) \% 11 = 35 \% 11 = 2$$

依此类推。

如你所见，不知道公式，每个连续的值似乎都是随机的。这正是称为伪随机数的原因。这个函数不是很好的伪随机数生成器，原因是它生成的数字很快就会开始重复。取决于库的实现，C++的伪随机数生成器各不相同。但基本思路就是这样，只是进行了一些增强，使随机数序列不至于这么快重复。

从不同的种子值开始，将得到一组不同的随机数。本例的种子值总是 1。然而，将不断变化的数字作为种子值，比如计算机时钟报告的时间，那么每次运行程序，都可能得到一组不同的随机数。

预定义函数 srand 用于为 C++随机数生成器提供种子值。函数不返回值，获取一个无符号整数作为种子值。例如，要将值 35 作为固定种子值，可以执行：

```
srand(35);
```

要每次都改变随机数序列，可以使用当前时间作为种子值。在大多数系统上，预定义函数 time(0)都返回自 1970 年 1 月 1 日以来经过的秒数[①]。time 函数要求包含 ctime 库。

```
#include <cstdlib>
#include <ctime>
...
srand(time(0));
```

获得随机数则要调用 rand 函数，它返回 0～RAND_MAX 的一个整数。RAND_MAX 是 cstdlib 中定义的常量，保证至少等于 32767。一般需要的不是 0～RAND_MAX 的数，所以要执行求余和加法运算对这个数进行比例缩放。例如，模拟掷六面骰子可以用以下语句：

```
int die = (rand() % 6) + 1;
```

随机数被 6 除的余数在 0～5 之间。再加 1，就得到了 1～6 的随机数。

务必注意，只向随机数生成器提供一次种子值。一个常见的错误是每次生成随机数都调用 srand。将 srand 和 rand 放到循环中，结果是可能生成一组相同的数字，因为计算机的运行速度很快，造成多次调用 srand 时获得的时间值没有变化。

强制类型转换

记住，9/2 是整数除法，结果是 4 而不是 4.5。如果希望除法运算生成 double 类型的数(也就是要保留小数部分)，那么在除法运算中的两个数字中，至少有一个必须是 double 类型。例如，9/2.0 的求值结果就是 4.5。执行除法运算时，只要其中一个数是常量，就可为此常量添加一个小数点和一个 0。这样一来，除法运算生成的值肯定会保留小数。

但是，假如除法运算中的两个操作数都是变量，会发生什么情况呢？如下所示：

```
int totalCandy, numberOfPeople;
double candyPerPerson;
<程序将 totalCandy 设为 9，将 numberOfPeople 设为 2，
    具体如何设置无关紧要>
candyPerPerson = totalCandy / numberOfPeople;
```

除非将 totalCandy 或 numberOfPeople 变量的值转换成 double 类型的值，否则除法运

① 自 1970 年 1 月 1 日以来经过的秒数称为 Unix 时间。

算的结果就是 4，而不是你所希望的 4.5。即使 candyPerPerson 变量是 double 类型，也于事无补。除法运算的结果值 4 虽然会先转换成 double 值，再存储到 candyPerPerson 变量中，但那时已经太迟了。4 转换为 4.0 后，candyPerPerson 的终值是 4.0，仍然不是 4.5。如果除法运算中的其中一个数是常量，就可为其添加一个小数点和一个 0，从而将常量转换为 double 类型。但在本例中，两个操作数都是变量，所以不能采用那种技术。幸运的是，有一个办法能将 int 类型转换成 double 类型，而且无论常量还是变量，都支持这一转换。

在 C++中，可指示计算机把 int 类型的值强制转换为 double 类型的值。例如，为了指示计算机“将值 9 转换为 double 类型”，可使用以下方式：

```
static_cast<double>(9)
```

static_cast<double>是一种预定义函数，它将某种类型的值(比如 9)转换成 double 类型(比如 9.0)。像 static_cast<double>(9)这样的表达式称为**强制类型转换**。可用其他变量或表达式代替其中的 9。除 double 外，还可使用其他类型名称，从而将某种类型转换为除了 double 之外的其他类型，但我们以后才会讲述这个主题。

例如，以下语句通过强制类型转换把数字 9 从 int 类型转换为 double 类型，使 answer 的值成为 4.5：

```
double answer;
    answer = static_cast<double>(9)/2;
```

将强制类型转换应用于常量(如 9)，可使程序更容易理解，因为它能更清楚地传达你的意图。但是，将强制类型转换应用于 int 类型的常量，发挥不出这种技术的真正优势。要将 9 转换为 double 类型，完全可以直接用 9.0 代替 static_cast<double>(9)。但是，如果除法运算仅涉及变量，就只能依靠强制类型转换了。使用强制类型转换，可以重写前一个例子，确保 candyPerPerson 获得正确的值 4.5(而不是 4.0)。为此，唯一需要修改的地方是用 static_cast<double>(totalCandy)代替 totalCandy。修改后的代码如下所示：

```
int totalCandy, numberOfPeople;
double candyPerPerson;
<程序将 totalCandy 设为 9，将 numberOfPeople 设为 2，
    具体如何设置无关紧要>
candyPerPerson =
    static_cast<double>(totalCandy)/numberOfPeople;
```

在以上代码使用的强制类型转换中，请注意圆括号的位置。应在执行除法运算之前执行强制类型转换，这样才能使除法操作符真正作用于 double 类型的值。如果在除法运算结束之后再执行转型，小数点之后的小数早已丢失了。如果将以上代码的最后一行错误地写为如下形式：

```
candyPerPerson =
static_cast<double>(totalCandy / numberOfPeople);// 错误!
```

那么 candyPerPerson 的值还是 4.0，而不是 4.5。

将 int 转型为 double 的函数

static_cast<double>可作为一个预定义函数使用，能将一种类型的值转换为 double 类型的值。例如，static_cast<double>(2)返回 2.0。这称为强制类型转换。利用该技术，还可将值转换成除 double 之外的其他类型，但本书前面只需执行到 double 的强制类型转换。

语法

```
static_cast<double>(Expression of Type int)
```

示例

```
int totalPot, numberOfWinners;
double yourWinnings;
. . .
yourWinnings = static_cast<double>(totalPot) / numberOfWinners;
```

强制类型转换的古老形式

如前一小节所述，static_cast<double>是执行强制类型转换的首选方式。但老版本 C++使用一种不同的方式表示强制类型转换。这种古老的形式直接使用类型名，如同它是函数名一样。所以，double(9)返回 9.0。如 candyPerPerson 是 double 变量，totalCandy 和 numberOfPeople 都是 int 变量，以下两个赋值语句就是等价的：

```
candyPerPerson =
      static_cast<double>(totalCandy)/numberOfPeople;
```

和

```
candyPerPerson =
      double(totalCandy)/numberOfPeople;
```

虽然 static_cast<**double**>(totalCandy)和 double(totalCandy)等价，但应该坚持使用前一种形式，因为在未来的 C++版本中，可能不再支持后一种。

陷阱：整数除法丢弃了小数部分

执行整数除法(比如 11/2)时，人们很容易忘记 11/2 的值是 5，而不是 5.5。结果是截去小数部分之后的整数。例如以下表达式：

```
double d;
d = 11/2;
```

这里的除法是整数除法，结果是 5，再将该值转换为 double 类型，最后赋给 d。小数部分会被完全丢弃。注意，虽然 d 被定义成 double 类型，但也无法改变除法结果是整数的事实。变量 d 获得的值是 5.0，而非 5.5。 ■

√ 自测题

1. 求以下算术表达式的值：

```
sqrt(16.0)  sqrt(16)         pow(2.0, 3.0)
pow(2, 3)   pow(2.0, 3)      pow(1.1, 2)
```

```
abs(3)      abs(-3)              abs(0)
fabs(-3.0)  fabs(-3.5)           fabs(3.5)
ceil(5.1)   ceil(5.8)            floor(5.1)
floor(5.8)  pow(3.0, 2)/2.0      pow(3.0, 2)/2
7/abs(-2)   (7 + sqrt(4.0))/3.0  sqrt(pow(3, 2))
```

2. 将以下数学公式转换为 C++算术表达式：

$$\sqrt{x+y} \quad x^{y+7} \quad \sqrt{area+fudge} \quad \sqrt{\frac{time+tide}{nobody}} \quad \frac{-b+\sqrt{b^2-4ac}}{2a} \quad |x-y|$$

3. 写完整 C++程序计算并输出圆周率 PI 的平方根。PI 近似等于 3.14159。`const double PI` 已在 `cmath` 中预定义，鼓励使用该预定义常量。

4. 编写并编译一些小程序来测试以下问题。
   a. 判断你的编译器是允许将`#include <iostream>`放在一行中的任意位置，还是要求`#`必须行首顶格。
   b. 判断你的编译器是否允许`#`和 `include` 之间存在空格。

4.3　程序员自定义函数

定做的西装肯定比大批量生产的好。

——*我的裁缝叔叔*

上一节描述了如何使用预定义函数，本节描述如何定义自己的函数。

函数定义

可以自己定义函数，要么和 `main` 部分一起放到同一个文件中，要么放到一个单独的文件中，使函数能由几个不同的程序使用。在这两种情况下，函数定义方式是一样的。但就目前来说，我们假定函数定义和程序 `main` 部分在同一个文件中。

图 4.3 在一个完整的程序中包含了一个示范函数定义，并演示了对这个函数的调用。函数名为 `totalCost`，它获取两个实参：商品价格和商品数量。函数返回含有销售税的总价。函数的调用方式与预定义函数一样。但是，函数的描述(这是程序员必须写的)有点儿复杂。

图 4.3　函数的定义

```
1  #include <iostream>
2  using namespace std;
3
4  double totalCost(int numberPar, double pricePar);      ←——— 函数声明
5  // 计算以单价 pricePar 购买 numberPar 件商品的总价，含 5%销售税
6
7
8  int main()
9  {
10     double price, bill;
11     int number;
12
13     cout << "Enter the number of items purchased: ";
14     cin >> number;
15     cout << "Enter the price per item $";
16     cin >> price;
```

```
17
18      bill = totalCost(number, price);
19
20      cout.setf(ios::fixed);
21      cout.setf(ios::showpoint);
22      cout.precision(2);
23      cout << number << " items at "
24           << "$" << price << " each.\n"
25           << "Final bill, including tax, is $" << bill
26           << endl;
27
28      return 0;
29  }
30
31  double totalCost(int numberPar, double pricePar)
32  {
33      const double TAX_RATE = 0.05; //5%的销售税
34      double subtotal;
35
36      subtotal = pricePar * numberPar;
37      return (subtotal + subtotal * TAX_RATE);
38  }
```

函数调用

函数头

函数主体

函数定义

示范对话

```
Enter the number of items purchased: 2
Enter the price per item $10.10
2 items at $10.10 each.
Final bill, including tax, is $21.21
```

函数描述分为两部分，即函数声明和函数定义。其中，**函数声明**也称为**函数原型**，描述了如何调用函数。C++要求在调用函数之前，要么存在完整函数定义，要么存在函数声明。图 4.3 将 totalCost 的函数声明放在较靠前的位置，如下所示：

```
double totalCost(int numberPar, double pricePar);
```

函数声明指出为了调用函数所需知道的一切，包括函数名称(本例是 totalCost)，函数需要多少个实参，以及这些实参是什么类型(本例要获取两个实参，第一个是 int 类型，第二个是 double 类型)。标识符 numberPar 和 pricePar 是形参。**形参**用作占位符，它在那个位置等待着实参(调用函数时实际传递的参数值)。由于写函数声明时还无法确定实参，所以用形参代替实参。形参名称可以是任意合法的标识符，但我们暂时在形参名称末尾添加 Par，这样可以更容易地将它们与程序中的其他项目区分。注意，函数声明以分号结尾。

函数声明的第一个单词指定了函数的返回值类型。因此，totalCost 函数的返回值类型是 double。

可以看出，图 4.3 的函数调用完全符合函数声明的各项要求。在以下语句中：

```
bill = totalCost(number, price);
```

等号右侧的表达式就是函数调用。函数名称是 totalCost，向它传递了两个实参：第一个是 int 类型，第二个是 double 类型。bill 变量是 double 类型，正好可以接收函数返回的 double 值。所有细节都是由函数声明确定好的。

编译器不关心函数声明是否带注释，但作为好习惯，应坚持用注释解释函数的返回值。

函 数 声 明

函数声明指出写函数调用所需的全部细节。函数声明和函数定义至少要有一样放到函数调用之前。函数声明通常放在程序的 main 部分之前。

语法

```
Type_Returned Function_Name(Parameter_List);     ← 不要忘记添加这个分号
Function Declaration Comment
```

其中，*Parameter_List* 是一组以逗号分隔的参数(形参)：

```
Type 1 Formal Parameter 1, Type 2 Formal Parameter 2, ...
              ..., Type Last Formal Parameter Last
```

示例

```
double totalWeight(int number, double weightOfOne);
//返回单位重量为weightOfOne 的number 件商品的总重量
```

图 4.3 中，函数定义被放到最底部。**函数定义**描述该函数如何计算返回值。将函数视为程序中的一个小程序，函数定义好比这个小程序的代码。事实上，函数定义的语法与程序 main 部分相似。函数定义包括函数头，后跟函数主体。**函数头**写法与函数声明一样，只是末尾没有分号。这使函数头显得有点儿重复，但习惯了就好。

虽然函数声明告诉你写函数调用所需的一切，但不能告诉你具体的返回值。具体返回值由函数主体确定。**函数主体**跟在函数头之后，用于完成函数定义。函数主体由声明和可执行语句构成，并统一放到花括号内。所以，函数主体类似于程序 main 部分的主体。调用函数时，会在形参的位置“代入”实参，然后执行函数主体的可执行语句。返回值在执行 return 语句时确定(“代入”实参的细节将在本节后面详细讨论)。

return 语句由关键字 return 和随后的表达式构成。图 4.3 中的函数定义包含以下 return 语句：

```
return (subtotal + subtotal * TAX_RATE);
```

执行该语句时，以下表达式的值作为函数调用的值返回：

```
(subtotal + subtotal * TAX_RATE);
```

圆括号并非必需的。如果把 return 语句写成以下形式：

```
return subtotal + subtotal * TAX_RATE;
```

程序的运行和以前完全一样。但是，针对较长的表达式，圆括号可以使 return 语句更容易阅读。考虑到一致性，有的程序员建议总是使用圆括号，即使是简单的表达式。在图 4.3 的函数定义中，return 语句之后没有别的语句；即使有，也不会执行。执行 return 语句后，函数调用便终止了。

在图 4.3 中执行以下函数调用时具体会发生什么事情？

```
bill = totalCost(number, price);
```

首先在形参位置代入实参 number 和 price 的值，换言之，实参 number 和 price 的值会替换形参 numberPar 和 pricePar。在示范对话中，number 获得值 2，price 获得值

10.10。所以，2 和 10.10 分别替换形参 numberPar 和 pricePar。这个替换过程称为**传值调用**，形参通常称为**传值调用形参**(简称**传值参数**)。对于这个替换过程，有三点要注意。

其一，在形参位置代入的是实参的值。如果实参是变量，就代入变量的值，而不是代入变量本身。

其二，在第一个形参位置代入第一个实参，在第二个形参位置代入第二个实参，依此类推。

其三，在形参位置代入实参时(比如在 numberPar 位置代入 2)，该形参在函数主体中的所有实例都会被替换成实参的值(例如，函数主体中出现的所有 numberPar 都被替换成 2)。

图 4.4 总结了图 4.3 的函数调用过程。

图 4.4　函数调用详解

解析图 4.3 的函数调用

0. 调用函数前，变量 number 和 price 的值由 cin 语句分别设为 2 和 10.10(参见图 4.3 中的示范对话)。

1. 开始执行包含一个函数调用的语句，这个语句如下：

```
bill = totalCost(number, price);
```

2. number 的值(也就是 2)在形参 numberPar 位置代入，而 price 的值(也就是 10.10)在形参 pricePar 位置代入：

```
double totalCost(int numberPar, double pricePar)
{
    const double TAX_RATE = 0.05; // 5%的销售税
    double subtotal;

    subtotal = pricePar * numberPar;
    return (subtotal + subtotal*TAX_RATE);
}
```

在此位置代入 number 的值

在此位置代入 price 的值

所以上述代码变成如下形式：

```
double totalCost(int 2, double 10.10)
{
    const double TAX_RATE = 0.05; // 5%的销售税
    double subtotal;

    subtotal = 10.10 * 2;
    return (subtotal + subtotal*TAX_RATE);
}
```

3. 执行函数主体，也就是执行以下语句：

```
{
    const double TAX_RATE = 0.05; // 5%的销售税
    double subtotal;
    subtotal = 10.10 * 2;
    return (subtotal + subtotal * TAX_RATE);
}
```

4. 执行 return 语句时，return 之后的表达式的值会由函数返回。就本例来说，一旦执行以下语句：

```
return(subtotal + subtotal * TAX_RATE);
```

以下函数调用就会返回 (subtotal + subtotal * TAX_RATE) 的值，也就是 21.21：

```
totalCost(number, price)
```

所以，一旦以下语句执行完毕，等号左侧的 bill 变量的值会被设为 21.21：

```
bill = totalCost(number, price);
```

函数如同小程序

为了理解函数，请牢记以下三点。

- 函数定义如同小程序，调用函数相当于运行这个“小程序”。
- 函数使用形参(而不是 cin)来输入。函数的实参就是实际输入的值，它们在形参位置代入。
- 函数(本章讨论的这一种)通常不向屏幕发送任何输出，但会将一种“输出”送还给程序。函数会返回一个值，该值好比函数的“输出”。函数进行这样的输出是用 return 语句，而不是用 cout 语句。

返回布尔值的函数

函数可以返回布尔值。可在布尔表达式中用这种函数控制 if-else 语句，控制循环语句，或者在允许布尔表达式的其他任何地方使用。种函数的返回类型是 bool。

只要函数能返回布尔值 true 或 false，在允许使用布尔表达式的任何地方，都能调用该函数。这通常能增强程序的可读性。通过函数声明，可将复杂的布尔表达式与一个有意义的名称联系起来，然后在 if-else 语句或者允许使用布尔表达式的其他任何地方使用这个名称。例如以下语句：

```
if (((rate >= 10) && (rate < 20)) || (rate == 0))
{
    …
}
```

可改写为以下形式：

```
if (appropriate(rate))
{
    ….
}
```

前提是定义好以下函数：

```
bool appropriate(int rate)
{
    return (((rate >= 10) && (rate < 20)) || (rate == 0));
}
```

另一种形式的函数声明

函数声明中不一定要列出形参名称，以下两个函数声明等价：

```
double totalCost(int numberPar, double pricePar);
double totalCost(int, double);
```

我们将坚持采用第一种形式，以便能在函数声明的注释中引用特定的形参。然而，在某些描述函数的手册中，经常会见到第二种形式。[①]

但是，这种替代形式只适合函数声明。函数头中必须列出形参名称。

① 为了将程序链接到函数库，C++唯一需要知道的就是函数名和形参类型。形参名称只对函数定义有用。然而，程序除了要和编译器沟通之外，还要和程序员沟通。使用程序员为函数中的数据分配的(有意义的)名称，有助于理解该函数。

陷阱：实参顺序错误

调用函数时，计算机在第一个形参位置代入第一个实参，在第二个形参位置代入第二个实参，依此类推。只是机械式地代入，不会考虑代入的数据是否合理。如果在函数调用中弄错了实参顺序，程序将无法做你希望做的事情。为了体会这个问题，下面研究一下图 4.5 的程序。该程序的作者在调用 grade 函数时，不小心弄反了实参的顺序。第 18 行的函数调用的正确写法是：

```
letterGrade = grade(score, needToPass);
```

这是程序唯一的错误；但正是由于这个疏忽，一个倒霉的学生会被错误地评为某门课不及格。grade 函数很简单，所以在测试程序时，程序员有可能发现这个错误。但如果 grade 是比较复杂的函数，错误就很难发现。

实参类型与形参不匹配，编译器可能报告一条警告信息。遗憾的是，并非所有编译器都会给出类似的警告。更有甚者，在图 4.5 的情况下，不会有任何编译器报告实参顺序有错，因为无论什么顺序，实参类型和形参类型都是匹配的。

图 4.5　实参顺序有错

```
1  // 判断用户的成绩，要么为 Pass(及格)，要么为 Fail(不及格)
2  #include <iostream>
3  using namespace std;
4
5  char grade(int received_par, int min_score_par);
6  // 如果 received_par 大于或等于 min_score_par，返回'P'表示及格;
7  // 否则返回'F'表示不及格
8
9  int main()
10 {
11     int score, needToPass;
12     char letterGrade;
13
14     cout << "Enter your score"
15          << " and the minimum needed to pass:\n";
16     cin >> score >> needToPass;
17
18     letterGrade = grade(needToPass, score);
19
20     cout << "You received a score of " << score << endl
21          << "Minimum to pass is " << needToPass << endl;
22
23     if (letterGrade == 'P')
24         cout << "You Passed. Congratulations!\n";
25     else
26         cout << "Sorry. You failed.\n";
27
28     cout << letterGrade
29          << " will be entered in your record.\n";
30
31     return 0;
32 }
33
34 char grade(int received_par, int min_score_par)
35 {
36     if (received_par >= min_score_par)
37         return 'P';
38     else
39         return 'F';
40 }
```

示范对话

```
Enter your score and the minimum needed to pass:
98 60
You received a score of 98
Minimum to pass is 60
Sorry. You failed.
F will be entered in your record.
```

函数定义语法小结

视频讲解：*Programmer-Defined Function Example*

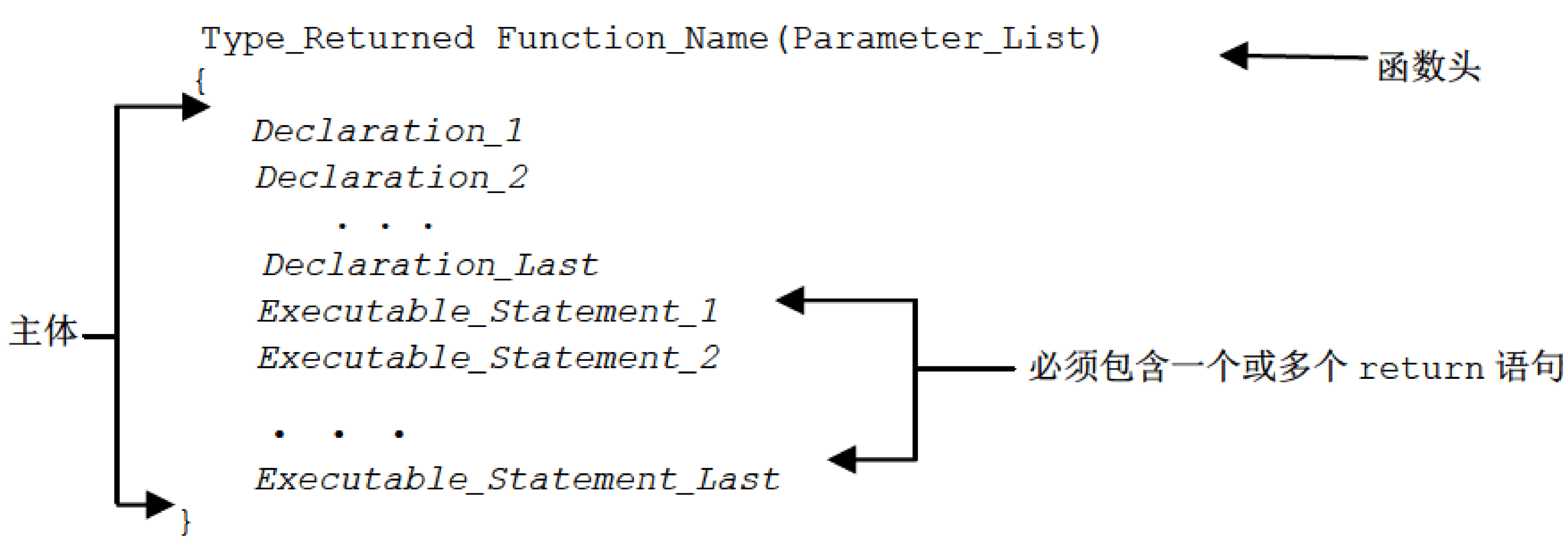
函数声明通常放到程序的 main 部分之前，函数定义通常放到程序的 main 部分之后(或者就像本书后面讲到的那样放到单独文件中)。图 4.6 总结了函数声明和函数定义的语法。但实际声明和定义函数时可以更灵活一些。在函数定义中，声明和可执行语句允许混合使用，只要每个变量在使用前声明。这和程序 main 部分是一样的规则。不过，除非有充分的理由，否则最好还是像图 4.6 那样，首先给出所有声明。

图 4.6 有返回值的函数的语法

函数声明

```
Type_Returned Function_Name(Parameter_List);
// Function_Declaration_Comment
```

函数定义



由于函数要在执行 return 语句时才返回值，所以在函数主体中，必须包含一个或多个 return 语句。函数定义允许包含多个 return 语句。例如，代码主体可能含有 if-else 语句，而 if-else 语句的每个分支都可能包含一个 return 语句，如图 4.5 所示。

函数定义中，任何合理的空白间隔和换行都是编译器允许的。但是，为函数定义使用的缩进和布局规则一定要和程序 main 部分一致。特别注意在函数定义和图 4.6 中的花括号的格式。标记函数主体开始和结束的花括号单独占一行，从而突出函数主体。

再论函数定义的位置

前面讨论了函数定义和函数声明通常应放在什么位置。一般情况下，这些是函数声明和定义的最佳位置。但编译器也允许将函数定义和声明放到程序中的其他位置。对于函数定义和声明的位置，一个更准确的规则是：在调用每个函数之前，都必须出现该函数的函

数声明或者该函数的定义。例如，假定将所有函数定义都放到程序的 main 部分之前，就不需要包含任何函数声明。知道这个更笼统的规则之后，就能理解其他书中的 C++程序。但就目前来说，建议像本书示范程序那样布局函数定义和函数声明。我们采用的风格将为你学习构建自己的函数库打下良好基础，同时也是大多数 C++程序员采用的风格。

编程提示：在分支语句中使用函数调用

switch 语句和多路 if-else 语句允许在每个分支都放几个不同的语句。但这会影响可读性。看看第 3 章图 3.7 的 switch 语句。与选项 1、2 和 3 对应的每个分支都可以是一个函数调用。这使 switch 语句的布局和程序的总体结构显得更清晰。如果将每个分支的每行代码都放到 switch 语句中，而不是用函数调用代替，那么 switch 会成为 C++语句的汪洋大海。事实上，switch 语句可能一屏都显示不完。

√ 自测题

5. 以下程序会输出什么结果？

```
#include <iostream>
using namespace std;
char mystery(int firstPar, int secondPar);
int main()
{
    cout << mystery(10, 9) << "ow\n";
    return 0;
}

char mystery(int firstPar, int secondPar)
{
    if (firstPar >= secondPar)
        return 'W';
    else
        return 'H';

}
```

6. 写一个函数的函数声明和函数定义。函数获取三个 int 类型的实参，返回这三个实参之和。
7. 写一个函数的函数声明和函数定义。函数获取一个 int 类型的实参和一个 double 类型的实参，返回这两个参数的平均值。
8. 写一个函数的函数声明和函数定义。函数获取一个 double 类型的实参。如果实参为正值，函数返回字符(char)值'P'；如果实参为零值或者负值，则返回'N'。
9. 详细解释传值调用机制。
10. 描述预定义函数(即库函数)和用户(程序员)自定义函数在用法上的异同。
11. 为名为 inOrder 的函数写函数定义。函数获取三个 int 类型的参数。如果三个参数按升序排序，函数返回 true；否则返回 false。例如，inOrder(1, 2, 3)和 inOrder(1, 2, 2)都返回 true，而 inOrder(1, 3, 2)返回 false。
12. 为名为 even 的函数写函数定义。函数获取一个 int 类型的参数，返回一个 bool 值。如果参数是偶数，函数返回 true；否则返回 false。
13. 为名为 isDigit 的函数写函数定义。函数获取一个 char 类型的参数，返回一个 bool 值。如果参数是一个十进制数位，函数返回 true；否则返回 false。
14. 为名为 isRootOf 的函数写函数定义。函数获取两个 int 类型的参数，返回一个 bool 值。如果第一个参数是第二个参数的平方根，函数返回 true；否则返回 false。

4.4　过 程 抽 象

原因虽未明，结果却昭然。

——*奥维德，《变形记》第四卷*

黑盒的比喻

使用程序的人没必要了解程序的具体编码。如果必须了解并记住自己使用的编译器的代码，可以想象难度有多大。每个程序都执行特定的任务，比如编译程序，或者检查论文中的单词拼写。只需知道程序能执行什么任务，不必(或者至少不需要)知道程序如何执行这个任务。函数好比小程序，应该能以类似的方式使用。在程序中使用函数时，程序员需要知道函数能够做什么(比如能计算平方根，能将温度从华氏度换算为摄氏度等)，但不需要知道函数如何完成这个任务。换言之，我们像对待黑盒那样对待函数。

说一样东西是**黑盒**，意思是说它就像一个物理设备。知道怎么使用，但内部工作原理是保密的，因为内部机构封闭在一个黑盒中，无法看到内部(也不能强行把它撬开)。良好设计的函数可作为黑盒使用。程序员只需知道一旦向其传递恰当的实参，就会返回恰当的返回值。有时将设计函数，使之能作为黑盒使用的过程称为**信息隐藏**，强调“函数主体在使用者面前隐藏”这一事实。

图 4.7 展示了 newBalance 函数的声明及其两个不同的定义。正如函数声明的注释所述，newBalance 函数用于计算在添加了单利之后，一个银行账户的新余额。例如，假定账户初始余额是 100 美元，为其添加 4.5%的利息，则新的余额为 104.50 美元。因此，以下代码将 vacationFund 的值从 100.00 变成 104.50：

```
vacationFund = 100.00
vacationFund = newBalance(vacationFund, 4.5);
```

图 4.7　等价于黑盒的函数定义

函数声明

```
1 double newBalance(double balancePar, double ratePar);
2 // 返回添加单利之后，银行账户的余额
3 // 形参 balancePar 是旧余额
4 // 形参 ratePar 是利率
5 // 例如，假定 ratePar 是 5.0，利率就是 5%，
6 // 所以 newBalance(100, 5.0)返回 105.00
```

定义 1

```
double newBalance(double balancePar, double ratePar)
{
    double interestFraction, interest;

    interestFraction = ratePar/100;
    interest = interestFraction*balancePar;
    return (balancePar + interest);
}
```

定义 2

```
double newBalance(double balancePar, double ratePar)
```

```
{
    double interestFraction, updatedBalance;

    interestFraction = ratePar/100;
    updatedBalance = balancePar*(1 + interestFraction);
    return updatedBalance;
}
```

程序员具体使用图 4.7 中的哪个 newBalance 实现是无关紧要的。两个定义生成的函数返回完全相同的值。这感觉就像是用一个黑盒将函数定义的主体包装起来，使程序员并不知道具体使用哪个实现。为了使用 newBalance 函数，程序员只需看一看函数声明和注释就好了。

函数作为黑盒来编写和使用，这也称为**过程抽象**。用 C++进行程序设计时，称为函数抽象似乎更恰当。但“过程”这个术语比“函数”还要常规。计算机科学工作者用“过程”一词表示所有“函数风格”的指令集，所以他们使用了“过程抽象”这一说法。“抽象”是指，将函数当作一个黑盒使用时，实际是对函数主体代码的细节进行提炼(抽象)。可将这种技术称为**黑盒原则**、**过程抽象原则**或者**信息隐藏**。三个术语的含义一样。不管怎样称呼，最重要的是在设计和编写函数定义时采用这种技术。

过 程 抽 象

应用于函数定义时，过程抽象原则是指你写的函数应该能像一个黑盒那样使用，也就是说，使用该函数的程序员不需要查看函数定义主体来了解函数的工作原理。程序员为了使用这个函数，只需查看函数声明及其注释。为了确保函数定义具有这一重要的属性，应严格遵守以下原则。

怎样写返回值的黑盒函数定义

- 在函数声明的注释中，指出对函数实参的所有要求，并指出用这些实参来调用函数，函数的返回值是什么。
- 函数主体使用的所有变量都应该在函数主体中声明(形参不需要声明，因为它们已在函数声明中列出)。

编程提示：选择形参名称

过程抽象原则表明，函数应该是自主性的模块，它独立于程序的其余部分进行设计。在大型编程项目中，可能指派另外的程序员编写每一个函数。程序员应该为形参选择最有意义的名称。用于替换形参的实参可能是程序的 main 部分中的变量。这些变量也应该使用有意义的名称，而且这些名称通常由其他人决定，而不是由写函数定义的程序员决定。这样一来，部分或全部实参可能与部分形参重名。但这完全能够接受。对于作为实参使用的变量，无论选择的是什么名称，都不会和形参冲突。毕竟，函数使用的只是实参的值。将变量作为函数实参使用时，函数只获取变量的值，而忽略变量名称。

现在已经知道，对形参名称的选择是完全自由的。所以，我们将停止在形参名称末尾添加 Par。例如，图 4.8 重写了图 4.3 的 totalCost 函数的定义，其中的形参命名为 number 和 price，而不是 numberPar 和 pricePar。用图 4.8 的版本替换图 4.3 中 totalCost 函

数声明和定义，程序将以完全相同的方式工作——即使形参名称已变为 number 和 price，而且程序 main 部分也有名为 number 和 price 的变量。

图 4.8　更简单的形参名

函数声明

```
1  double totalCost(int number, double price);
2  // 计算以单价price购买number件商品的总价，含5%的销售税
```

函数定义

```
1  double totalCost(int number, double price)
2  {
3      const double TAX_RATE = 0.05; // 5%的销售税
4      double subtotal;
5      subtotal = price * number;
6      return (subtotal + subtotal * TAX_RATE);
7  }
```

编程提示：嵌套循环

只要在代码中看到嵌套循环，就应考虑是否要运用过程抽象原则。以第 3 章图 3.15 的显式嵌套循环为例，它计算由所有动物保护者统计到的绿颈秃鹰蛋的总数。为了增强代码的可读性，可将循环转移到过程调用中，如图 4.9 所示。

对绿颈秃鹰的产卵数量进行汇总的两个版本是等价的。两个程序都生成相同的人机对话。但大多数人都会觉得图 4.9 的程序更易理解，因为循环主体是一个简单函数调用。观察外层循环，会认为对一个人的报告进行小计的操作是单个操作而非嵌套循环。

> **将循环主体变成函数调用**
>
> 任何时候要将一个循环嵌套到另一个循环中，或需在循环主体包括其他任何复杂计算，都考虑是否用一个函数调用代替。这样就可将复杂计算和程序其余部分分开，将编程任务分解成两个较小的子任务。

图 4.9　良好嵌套的循环

```
1  // 计算绿颈秃鹰产卵总数
2  // 由保护区的所有动物保护者统计
3  #include <iostream>
4  using namespace std;
5
6
7  int getOneTotal();
8  // 前条件：用户输入一组鸟蛋数量
9  // 最后以一个负数结束
10 // 后条件：总数等于所有鸟蛋计数的总和
11
12 int main()
13 {
14     cout << "This program tallies conservationist reports\n"
15          << "on the green-necked vulture.\n"
16          << "Each conservationist's report consists of\n"
17          << "a list of numbers. Each number is the count of\n"
18          << "the eggs observed in one"
19          << " green-necked vulture nest.\n"
20          << "This program then tallies"
```

```
21          << " the total number of eggs.\n";
22
23     int numberOfReports;
24     cout << "How many conservationist reports are there? ";
25     cin >> numberOfReports;
26
27     int grandTotal = 0, subtotal, count;
28     for (count = 1; count <= numberOfReports; count++)
29     {
30         cout << endl << "Enter the report of "
31              << "conservationist number " << count << endl;
32         subtotal = getOneTotal();
33         cout << "Total egg count for conservationist "
34              << " number " << count << " is "
35              << subtotal << endl;
36         grandTotal = grandTotal + subtotal;
37     }
38
39     cout << endl << "Total egg count for all reports = "
40          << grandTotal << endl;
41
42     return 0;
43 }
44
45
46 // 使用 iostream:
47 int getOneTotal()
48 {
49     int total;
50     cout << "Enter the number of eggs in each nest.\n"
51          << "Place a negative integer"
52          << " at the end of your list.\n";
53
54     total = 0;
55     int next;
56     cin >> next;
57     while (next >= 0)
58     {
59         total = total + next;
60         cin >> next;
61     }
62     return total;
63 }
```

示范对话

```
This program tallies conservationist reports
on the green-necked vulture.
Each conservationist's report consists of
a list of numbers. Each number is the count of
the eggs observed in one green-necked vulture nest.
This program then tallies the total number of eggs.
How many conservationist reports are there? 3

Enter the report of conservationist number 1
Enter the number of eggs in each nest.
Place a negative integer at the end of your list.
1 0 0 2 -1
Total egg count for conservationist number 1 is 3

Enter the report of conservationist number 2
Enter the number of eggs in each nest.
Place a negative integer at the end of your list.
0 3 1 -1
Total egg count for conservationist number 2 is 4

Enter the report of conservationist number 3
Enter the number of eggs in each nest.
Place a negative integer at the end of your list.
-1
Total egg count for conservationist number 3 is 0
Total egg count for all reports = 7
```

案例分析：购买比萨

对一些商品来说，大并不一定代表“性价比”高。买比萨时尤其要注意。比萨的尺寸是指直径，单位是英寸。但比萨真正的量要按面积来定(在厚度一致的情况下)，而面积与直径不成正比。对于一块 10 英寸的比萨和一块 12 英寸的比萨，大多数人可能无法迅速计算出两者在面积上的差异，从而不能快速判断哪种尺寸最适合购买——也就是说，哪种尺寸的比萨具有最低的“每平方英寸价格”；或者说，具有最高的“性价比”。在这个案例分析中，要设计程序来比较两种尺寸的比萨，确定哪一种最适合购买。

1. 问题定义

程序的输入和输出规范如下。

- 输入：输入两种尺寸的比萨的直径(以英寸为单位)和价格。
- 输出：输出两种尺寸的比萨各自的每平方英寸价格，并指出哪一种更适合购买，换言之，哪一种的每平方英寸价格最低(如果两者每平方英寸的价格一样，就认为尺寸较小的比萨性价比更高)。

2. 问题分析

我们将使用自顶向下设计法，将程序要解决的任务划分为以下子任务。

子任务 1：获得小比萨和大比萨的输入数据

子任务 2：计算小比萨每平方英寸的价格

子任务 3：计算大比萨每平方英寸的价格

子任务 4：判断哪一种更适合购买

子任务 5：输出结果

注意子任务 2 和 3。它们有两个重要的特点。

(1) 它们几乎是完全相同的任务。唯一的区别在于要用不同的数据来计算。子任务 2 和子任务 3 唯一要发生变化的就是比萨的尺寸和价格。

(2) 子任务 2 的结果和子任务 3 的结果都是单独一个值，即比萨每平方英寸的价格。

任何时候，只要一个子任务获取某些值(比如一些数字)，并返回单独一个值，就可以非常自然地将子任务作为一个函数来实现。假如两个或者更多的子任务执行相同的任务，就可作为相同的函数来实现该任务，只是每次使用时，用不同的实参来调用函数。因此，我们决定使用一个名为 unitPrice(单位价格)的函数，用它计算比萨每平方英寸的价格。函数声明和解释性注释如下：

```
double unitPrice(int diameter, double price);
// 返回比萨每平方英寸的价格。形参 diameter 以英寸为单位
// 形参 price 是比萨的价格
```

3. 算法设计

子任务 1 非常简单。程序要求输入值，并将输入的值存储到 4 个变量中，我们把这些变量分别命名为 diameterSmall，diameterLarge，priceSmall 和 priceLarge。

子任务 4 是一个简单例程。为了确定哪种比萨更适合购买，我们只需使用小于操作符

来比较两种比萨的每平方英寸价格。子任务 5 是一个简单的结果输出例程。

子任务 2 和子任务 3 实现成对 unitPrice 函数的调用。接着，要为这个函数设计一个算法。算法的难点是确定比萨的面积。一旦知道面积，就可执行除法，轻松地计算每平方英寸的价格，如下所示：

```
price/area
```

其中，area 是保存比萨面积的变量。以上表达式是 unitPrice 函数的返回值。但是，我们仍有必要制定一个用于计算比萨面积的方法。

比萨基本上是一个圆(由面包、奶酪、番茄酱和其他辅料组成)。圆的面积(也就是比萨的面积)计算公式为πr^2。其中，r 是圆的半径，π是圆周率，近似值为 3.14159。圆的半径是直径的一半。

unitPrice 函数的算法可概括如下。

unitPrice 函数算法概要

(1) 计算比萨半径。
(2) 使用圆的面积公式πr^2 计算比萨面积。
(3) 返回表达式(price/area)的值。

为了将以上算法概要转换成 C++代码，需要继续对它进行细化。我们准备使用伪代码描述这个算法的更详细的版本。**伪代码**是 C++代码与标准英文/中文文本的混合形式。使用伪代码，可使算法更精确，同时不必关心 C++语法的细节。以后，就可以轻松将伪代码转换为 C++代码。在我们的伪代码中，radius 和 area 是变量，分别保存比萨的半径和面积。

unitPrice 函数的伪代码

```
radius = diameter 的一半;
area = π * radius * radius;
return (price/area);
```

这样便结束了 unitPrice 函数的算法设计。下面将子任务 1～5 的解决方案转换成完整的 C++程序。

4．编码

子任务 1 的编码很简单，所以接下来要考虑子任务 2 和子任务 3。通过以下两个 unitPrice 函数调用，程序就能实现子任务 2 和子任务 3：

```
unitPriceSmall = unitPrice(diameterSmall, priceSmall);
unitPriceLarge = unitPrice(diameterLarge, priceLarge);
```

其中，unitPriceSmall 和 unitPriceLarge 是两个 double 类型的变量。函数定义的一个好处是可以在程序中多次调用那个函数。这便免去了重复相同(或者几乎相同)代码的麻烦。但在这里，仍然必须写出 unitPrice 函数的代码。

将以上伪代码转换成 C++代码，就得到了以下 unitPrice 函数主体：

```
{// unitPrice 函数主体的初稿
    const double PI = 3.14159;
    double radius, area;
```

```
    radius = diameter/2;
    area = PI * radius * radius;
    return (price/area);
}
```

注意，我们用修饰符 const 把 PI 声明为命名常量。同时，在上述代码中注意下面这一行代码：

```
radius = diameter/2;
```

这只是一个简单的除以 2 的计算，你可能认为这样写很正常。然而，如果真的这样写，这行代码就会包含一个严重的错误。我们希望除法运算得出比萨的半径，其中必须包含小数部分。例如，假定凑巧想买一种 13 英寸的比萨，则半径应该是 6.5 英寸。但是，diameter 变量是 int 类型，常量 2 也是 int 类型。因此，如第 2 章所述，这行代码会执行整数除法，13/2 的求值结果等于 6，而不是你希望的 6.5，这样一来，比萨半径就少了 0.5 英寸。如果不留意，这个问题很容易被忽略，结果可能是数百万“比萨消费者协会”[①]会员浪费他们的金钱购买尺寸有误的比萨。虽然就本程序而言，这个问题不太可能造成全球性经济衰退，但最起码，程序不能完成它的既定任务，即协助消费者选择更值得购买的比萨。在更重要的程序中，这种简单的错误可能造成灾难性的后果。

如何改正这个错误？我们希望“除以 2”是一个正规的除法运算，结果中的小数应予以保留。这种形式的除法要求除法操作符中的至少一个实参(操作数)为 double 类型。可以使用强制类型转换将常量 2 转换为 double 类型的一个值。以前说过，使用强制类型转换语句 static_cast<double>(2)，即可将 int 值 2 转换为 double 类型的值。因此，如果将上例的 2 替换成 static_cast<double>(2)，就能把除法操作符的第二个实参从 int 类型变成 double 类型。这样，除法运算就能生成我们希望的结果。重写的赋值语句如下所示：

```
radius = diameter/static_cast<double>(2);
```

图 4.10 是 unitPrice 函数的完整的、改正过的代码以及程序的其他部分。

图 4.10　购买比萨

```
1  // 判断两种比萨哪一种最适合购买
2  #include <iostream>
3  using namespace std;
4
5  double unitPrice(int diameter, double price);
6  // 返回比萨每平方英寸的价格
7  // 形参 diameter 是以英寸为单位
8  // 的比萨直径，形参 price 是比萨的价格
9
10 int main()
11 {
12     int diameterSmall, diameterLarge;
13     double priceSmall, unitPriceSmall,
14            priceLarge, unitPriceLarge;
15
16     cout << "Welcome to the Pizza Consumers Union.\n";
17     cout << "Enter diameter of a small pizza (in inches): ";
18     cin >> diameterSmall;
19     cout << "Enter the price of a small pizza: $";
20     cin >> priceSmall;
21     cout << "Enter diameter of a large pizza (in inches): ";
22     cin >> diameterLarge;
```

① Pizza Consumers Union，这是本书为了方便写程序而杜撰的一个协会。——译注

```
23      cout << "Enter the price of a large pizza: $";
24      cin >> priceLarge;
25
26      unitPriceSmall = unitPrice(diameterSmall, priceSmall);
27      unitPriceLarge = unitPrice(diameterLarge, priceLarge);
28
29      cout.setf(ios::fixed);
30      cout.setf(ios::showpoint);
31      cout.precision(2);
32      cout << "Small pizza:\n"
33           << "Diameter = " << diameterSmall << " inches\n"
34           << "Price = $" << priceSmall
35           << " Per square inch = $" << unitPriceSmall << endl
36           << "Large pizza:\n"
37           << "Diameter = " << diameterLarge << " inches\n"
38           << "Price = $" << priceLarge
39           << " Per square inch = $" << unitPriceLarge << endl;
40      if (unitPriceLarge < unitPriceSmall)
41         cout << "The large one is the better buy.\n";
42      else
43         cout << "The small one is the better buy.\n";
44
45      cout << "Buon Appetito!\n"; // 意大利语："祝你有个好胃口"
46        return 0;
47  }
48
49  double unitPrice(int diameter, double price)
50  {
51      const double PI = 3.14159;
52      double radius, area;
53
54      radius = diameter/static_cast<double>(2);
55      area = PI * radius * radius;
56      return (price/area);
57  }
58
```

示范对话

```
Welcome to the Pizza Consumers Union.
Enter diameter of a small pizza (in inches): 10
Enter the price of a small pizza: $7.50
Enter diameter of a large pizza (in inches): 13
Enter the price of a large pizza: $14.75
Small pizza:
Diameter = 10 inches
Price = $7.50 Per square inch = $0.10
Large pizza:
Diameter = 13 inches
Price = $14.75 Per square inch = $0.11
The small one is the better buy.
Buon Appetito!
```

由于 static_cast<double>(2) 的返回值是 2.0，所以可用常量 2.0 代替 static_cast<double>(2)。无论哪种方式，unitPrice 函数都会返回正确的值。但使用 static_cast<double>(2) 可以更明确地表示希望在除法运算结果中保留小数部分。换用常量 2.0，在修订或复制代码时，很容易将 2.0 误改为 2，造成不易发现的错误。

关于程序编码，还有另一个问题需要注意。如图 4.10 所示，在编写子任务 4 和子任务 5 的代码时，我们把这两个任务合并成一个代码小节，其中包括一组 cout 语句，再后跟一个 if-else 语句。某些情况下，如果两个任务非常简单，而且关系密切，可以考虑把它们合并成一个任务。

5．程序测试

不能仅仅因为一个程序通过编译，而且生成了看起来正确的结果，就判定程序是正确的。为了增强你对自己写的程序的信心，应该拿一些已通过其他方式(比如使用纸笔或者计算器)知道了正确答案的输入值来测试它。例如，买 2 英寸的比萨没什么意义，但仍可把它作为该程序的一个简单测试情况。之所以说它简单，因为很容易手动计算出答案。假定一个售价为 3.14 美元、直径为 2 英寸的比萨，我们要计算它的每平方英寸价格。由于直径是 2 英寸，所以半径是 1 英寸。半径为 1 英寸的比萨面积为 $3.14159 * 1^2$，也就是 3.14159。如果将这个数除以 3.14 美元，可得出该比萨每平方英寸的价格为 3.14/3.14159，即大约 1 美元。虽然对于比萨来说，这是一个十分荒谬的尺寸，而且对于如此小的比萨，也是一个十分荒谬的价格，但是，这样确实很容易判断 `unitPrice` 函数在获取这些实参之后应该返回什么值。

在这个简单情况下测试，可在一定程度上增强对程序的信心。但它仍然无法保证程序的正确性。不正确的程序偶尔也能给出正确答案，虽然它会在输入其他一些数据时给出错误的答案。某些情况下，错误的程序可能凑巧给出正确的输出，而你可能恰恰选中了这样的情况来测试。例如，假定没有纠正刚才编写 `unitPrice` 函数时发现的错误，在以下语句中仍然使用 `2`，而不是使用 `static_cast<double>(2)`：

```
radius = diameter/static_cast<double>(2);
```

也就是说，那一行代码仍然是：

```
radius = diameter/2;
```

那么，只要比萨直径是偶数，比如 `2`，`8`，`10` 或 `12`，程序给出的答案就会和除以 `static_cast<double>(2)` 时一样。换言之，程序表面上正确。虽然不大可能记住需要同时测试偶数和奇数尺寸的比萨，但只要用不同的比萨尺寸多测试几遍程序，就完全可能发现错误。

编程提示：使用伪代码

算法一般用伪代码表示。**伪代码**是 C++(或者其他编程语言)与普通英文(或者你使用的其他语言，比如中文)的混合体。使用伪代码能精确陈述算法，同时不需要纠缠于编程语言的语法。在算法陈述中，如果与一个步骤对应的 C++代码非常明确，就没必要再用英文或中文来陈述。相反，如果一个步骤很难用 C++来表示，那么用英文或中文来陈述能使算法更清晰。上个案例分析给出了伪代码的例子，我们用伪代码陈述 `unitPrice` 函数的算法。

√ 自测题

15. 为函数声明添加的注释有何用途？
16. 什么是应用于函数定义的过程抽象原则？
17. 我们说使用函数的程序员应该能将函数看成是一个黑盒，这具体是什么意思？
    提示：本题和上一题有密切联系。
18. 详细描述程序测试过程。

19. 研究一下 unitPrice 函数的两个定义。一个定义已在图 4.10 中给出。另一个定义基本相同，只是将强制类型转换 static_cast<double>(2)替换为常量 2.0。也就是说，以下语句：

```
radius = diameter/static_cast<double>(2);
```

要替换为以下形式：

```
radius = diameter/2.0;
```

从黑盒设计的角度出发，这两种函数定义是等价的吗？

4.5　作用域和局部变量

本地人，在外面可不出名。

——*俗语*

前一个小节建议像使用黑盒那样使用函数。为了定义能作为黑盒使用的函数，通常需要定义函数自己的、不受程序其他部分干扰的变量。“从属于”函数的变量称为该函数的局部变量。本节要介绍局部变量，并解释如何使用它们。

函数如同小程序

回顾前面的图 4.1，其中包含对预定义函数 sqrt 的调用。不需要为使用 sqrt 函数而了解它的函数定义细节。具体地说，不需要了解 sqrt 函数定义中声明了什么变量。你定义的函数也不例外。自定义函数中的变量声明独立于预定义函数中的变量声明。函数中的变量声明好比是另一个程序中的变量声明。在一个函数中声明了一个变量，再在程序的 main 部分(或其他函数主体)中声明同名变量，这两个变量将是不同的变量，即使它们重名了。下面研究一个程序，它的函数定义中的变量与程序中的另一个变量重名了。

图 4.11 的程序有两个名为 averagePea 的变量，一个在 estTotal 函数中声明和使用；另一个在程序的 main 部分中声明和使用。estTotal 函数定义的 averagePea 变量和程序 main 部分的 averagePea 变量是两个完全不同的变量。estTotal 函数好比是一个预定义函数。这两个名为 averagePea 的变量互不干扰，好像在两个完全不同的程序中。调用 estTotal 函数时，为变量 averagePea 赋了一个值，但这不会改变程序的 main 部分中的同名变量(对于图 4.11 的这个程序，这里只讲述与重名有关的细节，其他细节将在随后的 4.5 节进行讨论)。

在函数定义主体中声明的变量**局部于该函数**，或者说变量将函数作为**作用域**。在程序 main 主体中定义一个变量，就说该函数局部于程序的 main 部分，或者说变量将程序的 main 部分作为**作用域**。另外还有一些变量，它们不局部于任何函数或程序 main 部分。然而，我们不需要这些变量。我们使用的每个变量要么是一个函数定义的局部变量，要么是程序 main 部分的局部变量。如果说某个变量是**局部变量**，但既没有提到任何函数，也没有提到程序的 main 部分，就默认是函数的局部变量。

图 4.11　局部变量

```
1  // 计算一块豌豆试验田的平均收成
2  #include <iostream>
```

```
 3  using namespace std;
 4
 5  double estTotal(int minPeas, int maxPeas, int podCount);
 6  // 返回预计收获的豌豆总数
 7  // 形参 podCount 是豆荚数
 8  // 形参 minPeas 和 maxPeas 是每个豆荚中的最少和最多豌豆数
 9
10
11  int main()
12  {
13      int maxCount, minCount, podCount;
14      double averagePea, yield;
15
16      cout << "Enter minimum and maximum number of peas in a pod: ";
17      cin >> minCount >> maxCount;
18      cout << "Enter the number of pods: ";
19      cin >> podCount;
20      cout << "Enter the weight of an average pea (in ounces): ";
21      cin >> averagePea;
22
23      yield =
24           estTotal(minCount, maxCount, podCount) * averagePea;
25
26      cout.setf(ios::fixed);
27      cout.setf(ios::showpoint);
28      cout.precision(3);
29      cout << "Min number of peas per pod = " << minCount << endl
30           << "Max number of peas per pod = " << maxCount << endl
31           << "Pod count = " << podCount << endl
32           << "Average pea weight = "
33           << averagePea << " ounces" << endl
34           << "Estimated average yield = " << yield << " ounces"
35           << endl;
36
37      return 0;
38  }
39
40  double estTotal(int minPeas, int maxPeas, int podCount)
41  {
42      double averagePea;
43
44      averagePea = (maxPeas + minPeas)/2.0;
45      return (podCount * averagePea);
46  }
```

这个名为averagePea 的变量是程序main 部分的局部变量

这个名为 averagePea 的变量是函数 estTotal 的局部变量

示范对话

```
Enter minimum and maximum number of peas in a pod: 4 6
Enter the number of pods: 10
Enter the weight of an average pea (in ounces): 0.5
Min number of peas per pod = 4
Max number of peas per pod = 6
Pod count = 10
Average pea weight = 0.500 ounces
Estimated average yield = 25.000 ounces
```

局 部 变 量

在函数定义主体中声明变量，就说该变量局部于函数，或者说变量的**作用域**是那个函数。在程序 main 主体内定义变量，就说该函数局部于程序的 main 部分，或者说该变量的作用域是程序的 main 部分。

如果说某个变量是**局部变量**，但既没有提到任何函数，也没有提到程序的 main 部分，就默认表示该变量是某个函数定义的局部变量。如果变量是函数的局部变量，就可在程序的 main 部分或者其他函数定义中声明另一个同名变量。它们虽然同名，但却是不同的变量。

编程实例　豌豆试验田

图 4.11 的程序估算小花园的一小块豌豆试验田的总收成。estTotal 函数返回估计收获的豌豆总数。函数 estTotal 取 3 个实参。一个是收获的豆荚总数(podCount)。另两个估算一个豆荚的平均豌豆数。不同品种的豆荚包含不同数量的豌豆，所以另两个实参分别指定豆荚可能包含的最大和最小豌豆数(maxPeast 和 minPeas)。estTotal 函数求这两个数的平均值，该值就是豆荚的平均豌豆数。

全局常量和全局变量

第 2 章指出，可以(而且应该)使用 const 修饰符命名常量值。例如，图 4.10 通过以下声明为常量 3.14159 指定名称 PI：

```
const double PI = 3.14159;
```

在图 4.3 的以下声明中，使用 const 修饰符为销售税的税率指定了名称：

```
const double TAX_RATE = 0.05; // 5%的销售税
```

和变量声明一样，以前是将命名常量的声明放到使用它们的函数主体内。之所以这样做，是因为每个命名常量只由一个函数使用。但经常出现多个函数使用同一个命名常量的情况。此时，可将命名常量的声明放到程序较靠前的位置，在所有函数主体之外，也在程序 main 部分之外。这种命名常量称为**全局命名常量**。常量声明之后的所有函数定义都能使用。

图 4.12 的程序展示了全局命名常量的用法。该程序要求输入一个半径，根据半径计算圆的面积和球体的体积。程序作者事先找好了计算公式：

$$S = \pi \times r^2$$

$$V = (4/3) \times \pi \times r^3$$

两个公式都含有常量π，它的近似值为 3.14159。π是希腊字母，读作“pai”。在以前的程序中，我们已通过以下声明生成了名为 PI 的命名常量。将公式转换成 C++代码时可以使用该常量：

```
const double PI = 3.14159;
```

视频讲解：*Walkthrough of Functions and Local Variables*

图 4.12 的程序使用了相同的声明，但把它放到文件起始处，定义所有函数主体都能使用的全局命名常量。

图 4.12　全局命名常量

```
1  // 计算圆的面积和球体体积，
2  // 两个计算使用同样的半径
3  #include <iostream>
4  #include <cmath>
5  using namespace std;
6
7  const double PI = 3.14159;
8
9  double area(double radius);
10 // 返回具有指定半径的一个圆的面积
11
```

```
12  double volume(double radius);
13  // 返回具有指定半径的一个球体的体积
14
15  int main()
16  {
17     double radiusOfBoth, areaOfCircle, volumeOfSphere;
18
19     cout << "Enter a radius to use for both a circle\n"
20          << "and a sphere (in inches): ";
21     cin >> radiusOfBoth;
22
23     areaOfCircle = area(radiusOfBoth);
24     volumeOfSphere = volume(radiusOfBoth);
25
26     cout << "Radius = " << radiusOfBoth << " inches\n"
27          << "Area of circle = " << areaOfCircle
28          << " square inches\n" // 圆面积单位是平方英寸
29          << "Volume of sphere = " << volumeOfSphere
30          << " cubic inches\n"; // 球体体积单位是立方英寸
31
32     return 0;
33  }
34
35  double area(double radius)
36  {
37     return (PI * pow(radius, 2));
38  }
39
40  double volume(double radius)
41  {
42     return ((4.0/3.0) * PI * pow(radius, 3));
43  }
```

示范对话

```
Enter a radius to use for both a circle
and a sphere (in inches): 2
Radius = 2 inches
Area of circle = 12.5664 square inches
Volume of sphere = 33.5103 cubic inches
```

至于全局命名常量具体放到哪个位置，编译器提供了足够大的灵活性。但为了增强可读性，应该将所有 include 预编译指令放到一起，将所有全局命名常量声明放到一起，并将所有函数声明放到一起。我们将遵循约定俗成的做法，将所有全局命名常量声明放到 include 预编译指令之后、所有函数声明之前。

将所有命名常量声明放到程序起始处有利于可读性，即使只有一个函数使用该命名常量。命名常量在程序未来的版本中可能需要修改，所以放到程序起始处有利于查找。例如，将销售税税率常量声明放在财务程序的起始处，一旦以后提高税率，就可以轻松修改程序。

去掉 const 修饰符会将变量声明为**全局变量**。这种变量可由文件中的所有函数定义访问。但很少需要用到全局变量。另外，全局变量可能使程序难以理解和维护。所以我们不准备使用。积累了足够多的程序设计经验之后，才可以偶尔用一下。

传值形参是局部变量

形参是调用函数时要代入实参的空白位置。但形参的作用并非仅限于此。形参实际是局部于函数定义的变量，所以它们可以像函数定义中声明的任何局部变量那样使用。本章前面描述了传值调用机制，它在函数调用中用于处理实参。现在，我们可以更详细地定义这种“代入实参”机制。调用函数时，该函数的形参(局部变量)会被初始化为实参的值。

这是“在形参位置代入”的准确含义。通常，形参仅被用作一种空白位置(或称占位符)，要由相应实参的值来填充；但少数情况下，形参也可作为值会发生变化的变量使用。本节将提供一个将形参作为局部变量使用的例子。

图 4.13 展示的是 Dewey, Cheatham and Howe 律师事务所的计费程序。注意，和其他事务所不一样，Dewey, Cheatham and Howe 律师事务所虽然也是以一刻钟为单位收费，但不足一刻钟的部分是免费的。这正是称它为“一家有良心的律师事务所”[①]的原因。例如，假定他们为客户提供咨询服务的时间是 1 小时 14 分钟，则只收取 60 分钟(即 4 个一刻钟)的费用，而不像其他事务所那样收取 75 分钟(即 5 个一刻钟)的费用，这样算下来，客户只需支付 600 美元的咨询费。

图 4.13　形参用作局部变量

```
1   // 律师事务所计费程序
2   #include <iostream>
3   using namespace std;
4
5   const double RATE = 150.00; // 每刻钟收费金额
6
7   double fee(int hoursWorked, int minutesWorked);
8   //返回 hoursWorked 小时又 minutesWorked 分钟的法律服务费
9
10
11  int main()
12  {
13      int hours, minutes;
14      double bill;
15
16      cout << "Welcome to the offices of\n"
17           << "Dewey, Cheatham, and Howe.\n"
18           << "The law office with a heart.\n"
19           << "Enter the hours and minutes"
20           << " of your consultation:\n";
21      cin >> hours >> minutes;
22
23      bill = fee(hours, minutes);
24
25      cout.setf(ios::fixed);
26      cout.setf(ios::showpoint);
27      cout.precision(2);
28      cout << "For " << hours << " hours and " << minutes
29           << " minutes, your bill is $" << bill << endl;
30
31      return 0;
32  }
33
34  double fee(int hoursWorked, int minutesWorked)
35  {
36      int quarterHours;
37
38      minutesWorked = hoursWorked * 60 + minutesWorked;
39      quarterHours = minutesWorked/15;
40      return (quarterHours * RATE);
41  }
```

minutes 的值没有被 fee 函数调用更改

minutesWorked 是局部变量，已被初始化为 minutes 的值

示范对话

```
Welcome to the offices of
Dewey, Cheatham, and Howe.
The law office with a heart.
Enter the hours and minutes of your consultation:
2 45
For 2 hours and 45 minutes, your bill is $1650.00
```

注意 fee 函数定义中的形参 minutesWorked。它用作一个变量，其值由函数定义中的

① 即 The law office with a heart，参见图 4.13 的示范对话。——译注

下面这一行代码更改：

```
minutesWorked = hoursWorked * 60 + minutesWorked;
```

形参是局部变量，和函数主体中声明的其他变量一样。但是，不要为形参添加变量声明。在函数声明中列出形参 minutesWorked，相当于已经给出了变量声明。以下代码错误地开始 fee 函数定义，因其重复声明了 minutesWorked：

```
double fee(int hoursWorked, int minutesWorked)
{
   int quarterHours;
   int minutesWorked;          <------------ 不要这样做！
     . . .
```

块作用域

局部变量的作用域是指程序中能直接访问变量的那一部分，有时称为**局部作用域**。类似地，在程序开头、所有函数主体外部声明的全局标识符具有**全局作用域**。虽然存在差异，但局部和全局标识符实际是第 3 章描述的块作用域的例子。块(或代码块)是指花括号中的 C++代码，例外的是“全局块”，它是包含了全部代码的最外层的块。作用域规则指出，块中声明的标识符局部于那个块，只有从声明位置开始到块结束的代码才能访问。块通常嵌套。例如，main 函数的花括号定义了一个块，main 内的 for 循环定义的就是嵌套块。

图 4.14 的程序不计算任何实际的东西，只是演示了不同块中声明的标识符的作用域。在这个例子中，常量 GLOBAL_CONST 具有全局作用域，函数 function1 和 main 也是如此，因为它们在所有函数的外部声明。这就允许从 main 和 function1 中访问 GLOBAL_CONST。

图 4.14　局部、全局和块作用域

再论块作用域

```
1    #include <iostream>
2    using namespace std;
3
4    const double GLOBAL_CONST = 1.0;
5
6    int function1(int param);
7
8    int main()
9    {
10       int x;
11       double d = GLOBAL_CONST;
12
13       for (int i = 0; i < 10; i++)
14       {
15           x = function1(i);
16       }
17       return 0;
18   }
19
20   int function1(int param)
21   {
22       double y = GLOBAL_CONST;
23       ...
24       return 0;
25   }
```

局部和全局作用域是块作用域的例子。
变量只有在它的作用域中才能直接访问。

块作用域：变量 *i* 的作用域是 13-16行

局部于 *main*：*x* 的作用域是 10-18行，而变量 *d* 的作用域是 11-18行

全局作用域：常量 *GLOBAL_CONST* 的作用域是 4-25行，而函数 *function1* 的作用域是 6-25行

局部于 *function1*：变量 *param* 的作用域是 20-25 行，变量 *y* 的作用域是 22-25行

main 函数声明了变量 x 和 d，它们局部于 main，作用域到 main 块结束时为止。类似地，函数 function1 有一个 param 参数和局部变量 y，作用域到 function1 结束时为止。这两个变量都不能直接从其作用域的外部访问。局部变量和参数的作用域规则其实就是块

作用域规则，只是在这种情况下，块是声明变量或参数的那个函数。

图 4.14 的 for 循环展示了嵌套块的作用域。变量 i 在 for 循环中声明，所以只有到循环块结束时的作用域。在作用域外部的任何位置引用 i，即使仍然在 main 内部(比如在第 17 行)，都会造成编译错误。

可认为变量在作用域开始时创建，在作用域结束时销毁。例如，每次调用 function1，图 4.14 的局部变量 y 都会创建并初始化为 GLOBAL_CONST。如果第 23 行的代码更改 y 中存储的值，这些更改会在函数退出时丢失，而 y 会超出作用域，因为变量 y 会被销毁。重复调用 function1 不会记住 y 之前的值，而是会创建新 y。

除了块作用域，还有命名空间作用域和类作用域。类作用域在第 10 章讨论，命名空间作用域在第 12 章讨论。C++还定义了函数原型作用域，也就是函数原型中的参数的作用域。[①] 最后，C++支持用于标签的函数作用域。标签是 C 语言的遗留物，随同 goto 语句使用。一般不要使用，因为可能造成难以理清的逻辑。循环通过更容易理解的方式执行相同的任务。

再论命名空间

到目前为止，我们所有程序都从以下两行代码开始：

```
#include <iostream>
using namespace std;
```

但文件开头并非总是下面这行代码的最佳位置：

```
using namespace std;
```

终究会用到更多命名空间，而非仅仅是 std。事实上，甚至可能要在不同函数定义中使用不同命名空间。如果将以下指令放到函数定义主体的花括号内：

```
using namespace std;
```

using 指令就只应用于那个函数定义。这样就可在不同函数定义中使用不同命名空间，即使这些函数定义在同一个文件中，或者不同命名空间包含同名但不同含义的名称。

将 using 指令放到函数定义内，类似于将一个变量声明放到函数定义内。将变量定义放到函数定义内，该变量就局部于该函数；也就是说，变量声明的含义仅在函数定义的范围内有效。将 using 指令放到函数定义内，它就局部于函数定义；也就是说，using 指令的含义仅在函数定义的范围内有效。

本书要到稍晚的时候才会在 using 指令中使用除 std 之外的其他命名空间。但最好从现在起，就有意识地将 using 指令放到它们应该出现的位置。图 4.15 改写了图 4.12 的程序，这次将 using 指令放到它们应该出现的位置。图 4.15 的程序在行为上与图 4.12 的程序一样。就这个例子来说，区别只在于编程风格。但随着你开始使用更多的命名空间，就会发现它最终将影响程序的执行方式。

---

① 说的是函数声明(原型)而非函数定义。例如图 4.14 第 6 行，param 的作用域就在两个圆括号之间。第 20 行的 param 的作用域就是整个函数体了。——译注

图 4.15　使用命名空间

```
1  // 计算一个圆的面积和一个球体的体积
2  // 两个计算使用相同的半径
3  #include <iostream>
4  #include <cmath>
5
6  const double PI = 3.14159;
7
8  double area(double radius);
9  // 返回具有指定半径的一个圆的面积
10
11 double volume(double radius);
12 // 返回具有指定半径的一个球体的体积
13
14 int main()
15 {
16     using namespace std;
17
18     double radiusOfBoth, areaOfCircle, volumeOfSphere;
19
20     cout << "Enter a radius to use for both a circle\n"
21          << "and a sphere (in inches): ";
22     cin >> radiusOfBoth;
23
24     areaOfCircle = area(radiusOfBoth);
25     volumeOfSphere = volume(radiusOfBoth);
26
27     cout << "Radius = " << radiusOfBoth << " inches\n"
28          << "Area of circle = " << areaOfCircle
29          << " square inches\n"
30          << "Volume of sphere = " << volumeOfSphere
31          << " cubic inches\n";
32
33     return 0;
34 }
35
36
37 double area(double radius)
38 {
39     using namespace std;
40
41     return (PI * pow(radius, 2));
42 }
43
44 double volume(double radius)
45 {
46     using namespace std;
47
48     return ((4.0/3.0) * PI * pow(radius, 3));
49 }
```

该程序的示范对话与图 4.12 中的示范对话一样，故略去

√ 自测题

20. 要在函数定义中使用变量，应该在哪里声明它？在函数定义中，还是在程序的 main 部分中，还是哪里方便就放在哪里？

21. 假定 Function1 函数有一个 Sam 变量，它在 Function1 的定义中声明；Function2 函数也有一个 Sam 变量，它在 Function2 的定义中声明。假设其他一切正确，程序能通过编译吗？如果程序能编译，它能运行吗？如果能运行，会在运行时生成错误消息吗？如果既能运行，也不会在运行时报错，能给出正确输出吗？

22. 以下函数获取以英尺数和英寸数，换算为总英寸数后输出。例如，totalInches(1, 2)应该返回 14，因为 1 英尺 2 英寸等于 14 英寸。以下函数能满足这个设计要求吗？如果不能，请说明原因。

```
double totalInches(int feet, int inches)
{
    inches = 12 * feet + inches;
    return inches;
}
```

23. 为名为 readFilter 的函数编写函数声明和函数定义，该函数无参数，返回 double 值。函数提示用户输入一个 double 类型的值，将该值读入一个局部变量。如果读入的值大于或等于 0，函数就返回这个值；为负则返回 0。

编程实例　阶乘函数

图 4.16 包含阶乘函数的声明和定义。阶乘函数是常用的数学函数。在数学教科书中，阶乘函数通常写成 $n!$，定义为从 1 到 n 的所有整数的乘积。采用传统数学表示法，可将 $n!$ 定义为：

$$n! = 1\times2\times3\times\cdots\times n$$

函数定义用 while 循环执行乘法运算。注意，乘法运算的方向可能和你想象的相反。程序首先乘以 n，再乘以 $n-1$，再乘以 $n-2$，依此类推。

图 4.16　阶乘函数

函数声明

```
1  int factorial(int n);
2  // 返回 n 的阶乘。实参 n 应该是一个非负的整数
```

函数定义

```
1  int factorial(int n)
2  {
3      int product = 1;
4      while (n > 0)
5      {
6          product = n * product;
7          n--;           <-------- 形参 n 用作局部变量
8      }
9
10     return product;
11 }
```

factorial 函数定义使用了两个局部变量，其中一个是函数主体开始处声明的 product，另一个则是形参 n。由于形参是局部变量，所以我们可更改它的值。本例使用递减操作符--更改形参 n 的值(第 2 章已讨论了递减操作符)。

循环主体每次执行，变量 product 的值都会乘以 n 的值，然后通过 n--使 n 的值递减 1。调用函数时，如果将 3 作为实参传递给它，那么循环主体第一次执行完毕之后，product 的值是 3；循环主体第二次执行之后，product 的值 3*2；第三次是 3*2*1。然后，循环终止。因此，以下函数调用会把变量 x 设为 3*2*1，也就是 6：

```
x = factorial(3);
```

注意，局部变量 product 在声明的同时初始化为值 1(第 2 章讨论过如何在声明变量的同时初始化)。很容易看出，1 是变量 product 的正确初始值。为证实这一点，注意在第一次执行 while 循环主体之后，我们希望 product 的值等于形参 n 的原始值。如果 product 初始化为 1，就能满足这个要求。

4.6　重载函数名称

"……— 不过这说明有三百六十四天可以得到非生日礼物。"

"是的。"

"你知道，生日的礼物只有一天。这对你多光荣呀!"

"我不懂你所说的'光荣'是什么意思。"爱丽丝说。

矮胖子轻蔑地笑了："你当然不懂，等我告诉你。我的意思是你在争论中彻底失败了。"

"但是'光荣'的意思并不是'争论中彻底失败'呀。"爱丽丝反驳道。

"我用一个词，总是同我想要说的恰如其分的，既不重，也不轻。"矮胖子相当傲慢地说。

"问题是你怎么能造出一些词，它可以包含许多不同的意思呢？"

"问题是哪个是主宰——关键就在这里。"矮胖子说。

——刘易斯・卡洛尔，《爱丽丝镜中奇遇记》

C++允许为同一个函数名提供两个或更多的定义，这表示可以在不同情况下重用这些名称。例如，可以有三个名为 max 的函数：一个计算 2 个数的最大值；一个计算 3 个数的最大值；还有一个计算 4 个数的最大值。为同一个函数名称提供两个(或更多)函数定义，就称为**重载**函数名称。重载要求你更谨慎地定义函数。另外，除非能显著改善程序的可读性，否则最好不要重载。但是，如果条件合适，重载十分见效。

重载入门

假定要写计算两个数的平均值的程序，可使用以下函数定义：

```
double ave(double n1, double n2)
{
    return ((n1 + n2)/2.0);
}
```

假定还要用函数计算 3 个数的平均值，可以定义一个名为 ave3 的新函数：

```
double ave3(double n1, double n2, double n3)
{
    return ((n1 + n2 + n3)/3.0);
}
```

这样做是可行的。而且在许多编程语言中，这是唯一的选择。幸好，C++允许选择更优雅的方案。在 C++中，可以为两个函数使用同一个函数名称 ave。换言之，可以使用以下函数定义取代上面的 ave3 函数定义：

```
double ave(double n1, double n2, double n3)
{
    return ((n1 + n2 +n3)/3.0);
}
```

这样，ave 这个函数名就有了两个定义。这就是重载。重载的是函数名 ave。图 4.17 将 ave 的两个函数定义嵌入一个完整的示范程序。注意，每个函数定义都有自己的函数声明。

图 4.17　重载函数名称

```
1  //演示对函数名 ave 的重载
2  #include <iostream>
3
4  double ave(double n1, double n2);
5  // 返回两个数 n1 和 n2 的平均值
6
7  double ave(double n1, double n2, double n3);
8  // 返回三个数 n1，n2 和 n3 的平均值
9
10 int main()
11 {
12     using namespace std;
13     cout << "The average of 2.0, 2.5, and 3.0 is "
14          << ave(2.0, 2.5, 3.0) << endl;
15
16     cout << "The average of 4.5 and 5.5 is "
17          << ave(4.5, 5.5) << endl;
18
19     return 0;
20 }
21
22 double ave(double n1, double n2)          ←———— 两个实参
23 {
24     return ((n1 + n2)/2.0);
25 }
26
27 double ave(double n1, double n2, double n3)   ←———— 三个实参
28 {
29     return ((n1 + n2 + n3)/3.0);
30 }
31
32
```

输出

```
The average of 2.0, 2.5, and 3.0 is 2.50000
The average of 4.5 and 5.5 is 5.00000
```

重载是非常棒的概念。它使程序的可读性更高。它为程序员解决了必须绞尽脑汁为函数想名字的问题(一个本来很贴切的名字可能被其他函数占用了)。但是，当编译器遇到一个函数调用时，如果发现该函数名具有两个或者更多的定义，它怎么知道具体该用哪一个？编译器不可能读懂程序员的心思。为了弄清楚应该使用哪个函数定义，编译器会检查函数调用中的实参个数和实参类型。在图 4.17 的程序中，一个 ave 函数有两个实参，另一个则有 3 个。为确定要用哪一个定义，编译器只需统计函数调用中的实参个数。两个实参就用第一个定义，3 个实参用第二个。

只要为同一个函数名指定了两个或更多的函数定义，每个函数定义就必须有不同的实参规范；也就是说，任何两个同名的函数定义必须使用不同数量的形参，或者使用不同类型的形参(或同时满足这两个条件)。换言之，重载函数名时，函数声明的形参必须有区别。如果两个函数唯一的区别就是返回值的类型，则不能重载这个函数名。

你对重载其实并不陌生。第 2 章曾展示过重载除法操作符。两个操作数都是 int 类型，如 13/2，返回值就是整数除法结果(本例是 6)。而两个操作数如果一个或两个都是 double 类型，返回值就是标准除法所返回的值(例如，13/2.0 返回 6.5)。换言之，除法操作符 / 有两个定义，而且这两个定义不是依靠操作数的个数来区分的，而是依靠操作数的类型来

区分的。重载操作符/与重载函数名的区别在于，编译器已经完成了对操作符/的重载，而对函数名的重载要依靠你的程序进行。本书以后还会介绍如何重载操作符+和-等。

重载函数名

一个函数名具有两个或更多的函数定义称为**重载**。重载时，函数定义必须使用不同数量的形参，或者使用不同类型的形参(或同时满足这两个条件)。遇到一个函数调用时，编译器会选用形参数量及类型与传递的实参相匹配的函数定义。

编程实例　购买比萨(修订版)

利用图 4.10 的程序，比萨消费者协会获得了极大的成功。现在，每个人都总是选购最适合购买的比萨。有一家声誉不好的比萨店，它过去依靠欺骗消费者购买更贵的比萨，大发不义之财。但是，我们的程序终止了他们的恶行。现在，店主不思悔改，希望延续过去的卑鄙行径，并想出一个新花样来欺骗消费者。现在，他们同时提供圆形比萨和矩形比萨。他们知道我们写的那个程序不支持矩形比萨，所以希望能再次迷惑消费者。为此，需要更新程序以挫败他们的阴谋。下面要修改程序，使它能比较圆形比萨和矩形比萨的性价比。

这个比萨评估程序应该进行哪些修改，应该是一目了然的：需要稍微修改输入和输出，使其能处理两种不同形状的比萨；还要添加一个新函数，计算矩形比萨的每平方英寸价格。可在程序中使用以下函数定义来计算矩形比萨的每平方英寸价格：

```
double unitPriceRectangular
       (int length, int width, double price)
{
    double area = length * width;
    return (price/area);
}
```

但是，函数名实在太长，以至于要把函数头拆成两行。虽然这是有效的，但最好能用同一个名称(unitPrice)命名这两个函数，一个计算圆形比萨的每平方英寸价格，另一个计算矩形比萨的每平方英寸价格。由于 C++允许函数名重载，所以完全可以这样做。让 unitPrice 函数具有两个定义不会为编译器带来困扰，因为两者的实参个数不同。图 4.18 展示了购买比萨程序的修订版，它现在能对圆形和矩形比萨进行比较。

图 4.18　重载函数名

```
1  // 判断圆形比萨和矩形比萨，哪一种更适合购买
2  #include <iostream>
3
4  double unitPrice(int diameter, double price);
5  // 返回圆形比萨每平方英寸的价格。形参 diameter 是以英寸为单位
6  // 的比萨直径，形参 price 是比萨的价格
7
8
9  double unitPrice(int length, int width, double price);
10 // 返回矩形比萨每平方英寸的价格
11 // 比萨的长度和宽度分别是 length 英寸和 width 英寸
12 // 形参 price 是比萨的价格
13
14 int main()
```

```
15  {
16      using namespace std;
17      int diameter, length, width;
18      double priceRound, unitPriceRound,
19            priceRectangular, unitPriceRectangular;
20
21      cout << "Welcome to the Pizza Consumers Union.\n";
22      cout << "Enter the diameter in inches"
23           << " of a round pizza: ";
24      cin >> diameter;
25      cout << "Enter the price of a round pizza: $";
26      cin >> priceRound;
27      cout << "Enter length and width in inches\n"
28          << "of a rectangular pizza: ";
29      cin >> length >> width;
30      cout << "Enter the price of a rectangular pizza: $";
31      cin >> priceRectangular;
32
33      unitPriceRectangular =
34                unitPrice(length, width, priceRectangular);
35      unitPriceRound = unitPrice(diameter, priceRound);
36
37      cout.setf(ios::fixed);
38      cout.setf(ios::showpoint);
39      cout.precision(2);
40      cout << endl
41           << "Round pizza: Diameter = "
42           << diameter << " inches\n"
43           << "Price = $" << priceRound
44           << " Per square inch = $" << unitPriceRound
45           << endl
46           << "Rectangular pizza: Length = "
47           << length << " inches\n"
48           << "Rectangular pizza: Width = "
49           << width << " inches\n"
50           << "Price = $" << priceRectangular
51           << " Per square inch = $" << unitPriceRectangular
52           << endl;
53
54      if (unitPriceRound < unitPriceRectangular)
55         cout << "The round one is the better buy.\n";
56      else
57         cout << "The rectangular one is the better buy.\n";
58      cout << "Buon Appetito!\n";
59
60      return 0;
61  }
62
63  double unitPrice(int diameter, double price)
64  {
65      const double PI = 3.14159;
66      double radius, area;
67
68      radius = diameter/static_cast<double>(2);
69      area = PI * radius * radius;
70      return (price/area);
71  }
72
73  double unitPrice(int length, int width, double price)
74  {
75      double area = length * width;
76      return (price/area);
77  }
```

示范对话

```
Welcome to the Pizza Consumers Union.
Enter the diameter in inches of a round pizza: 10
Enter the price of a round pizza: $8.50
Enter length and width in inches
of a rectangular pizza: 6 4
Enter the price of a rectangular pizza: $7.55

Round pizza: Diameter = 10 inches
Price = $8.50 Per square inch = $0.11
Rectangular pizza: Length = 6 inches
Rectangular pizza: Width = 4 inches
Price = $7.55 Per square inch = $0.31
The round one is the better buy.
Buon Appetito!
```

自动类型转换

假定程序含有以下函数定义，而且没有重载函数名 mpg(换言之，它是名为 mpg 的函数的唯一定义)：

```
double mpg(double miles, double gallons)
// 返回每加仑汽油行驶的英里数(简称为 MPG)
{
    return (miles/gallons);
}
```

用 int 类型的实参调用 mpg 函数，C++自动将 int 类型的任何实参转换为 double 类型。因此，以下语句在屏幕上输出：22.5 miles per gallon：

```
cout << mpg(45, 2) << " miles per gallon";
```

C++将 45 转换为 45.0，将 2 转换为 2.0，再执行除法运算 45.0/2.0，获得返回值 22.5。

如果函数要求 double 类型的实参，但提供的是 int 类型的实参，C++会将 int 实参自动转换为 double 类型的值。这是一个很有用的转换，而且很自然，以至于我们很少深入地去想它。但是，重载可能干扰这种自动类型转换。下面研究一个例子。

假设你一时糊涂重载了函数名 mpg，程序中还包含以下 mpg 定义(除前面那个之外)：

```
int mpg(int goals, int misses)
// 返回 Measure of Perfect Goals(同样简称为 MPG)
// 计算公式为(goals - misses)
// goals 是进球数，misses 是失球数
{
    return (goals - misses);
}
```

在程序中同时包含函数名 mpg 的两个定义，以下语句会令人遗憾地输出：43 miles per gallon(因为 45 - 2 为 43)：

```
cout << mpg(45, 2) << " miles per gallon";
```

C++遇到函数调用 mpg(45, 2)时，由于该函数调用中有两个 int 类型的实参，所以 C++首先查看有两个 int 形参的 mpg 函数定义。如果找到匹配的函数定义，C++就使用该函数

定义。C++不会一开始就将 int 实参转换为 double 类型的值，除非只有那样才能找到匹配的函数定义。

mpg 的例子说明了重载的另外一个要点：不要为两个不相关的函数使用同一个函数名。函数名使用不当，肯定会导致混乱。

√ 自测题

24. 假设有两个函数定义，函数声明分别如下：

```
double score(double time, double distance);
int score(double points);
```

遇到以下函数调用时，会使用哪一个函数定义？为什么会使用它(x 是 double 类型)？

```
finalScore = score(x);
```

25. 假设有两个函数定义，函数声明分别如下：

```
double theAnswer(double data1, double data2);
double theAnswer(double time, int count);
```

遇到以下函数调用时，会使用哪一个函数定义？为什么会使用它(x 和 y 是 double 类型)？

```
x = theAnswer(y, 6.0);
```

26. 假设有两个函数定义，函数声明见自测题 25。遇到以下函数调用时，会使用哪一个函数定义？为什么会使用它？

```
x = theAnswer(5, 6);
```

27. 假设有两个函数定义，函数声明见自测题 25。遇到以下函数调用时，会使用哪一个函数定义？为什么会使用它？

```
x = theAnswer(5, 6.0);
```

28. 本题基于 4.6.2 节。那个总是企图欺骗消费者的比萨店店主打算推出一种正方形比萨。你能否重载 unitPrice 函数，使其除了能计算圆形比萨的每平方英寸价格，还能计算正方形比萨的每平方英寸价格？请说明原因。

29. 在图 4.18 的程序中，main 函数包含 using 指令：

```
using namespace std;
```

unitPrice 方法为什么不包含这个 using 指令？

小　结

- 设计算法最有效的手段是将任务分解成多个子任务，再将每个子任务分解成更小的子任务，依此类推。最终，子任务会变得非常小，很容易用 C++代码实现。这种设计方法学称为**自顶向下设计**。
- 返回值的函数好比小程序。传给函数的实参好比输入，返回值好比输出。
- 假如程序的某个子任务需要获取某些值作为输入，并生成一个值作为它唯一的结果，就可将该子任务作为函数实现。
- 函数应该能像黑盒那样使用。使用函数的程序员不需要知道函数的编码细节，只需知道函数声明以及对返回值进行描述的注释。这有时称为**过程抽象原则**。
- 函数定义中声明的变量局部于该函数，是函数的**局部变量**。
- 全局命名常量用修饰符 const 声明。全局命名常量的声明通常放到程序起始处，在 include 预编译指令之后，但在函数声明之前。
- 传值形参(本章唯一讨论的形参)是函数的局部变量。形参有时需要作为局部变量使用。
- 如果同一个函数名有两个或更多函数定义，表明函数被**重载**了。重载函数时，不同函数定义必须有不同数量的形参，或者有不同的形参类型(或同时满足这两个条件)。

自测题答案

1.
```
4.0     4.0     8.0
8.0     8.0     1.21
3       3       0
3.0     3.5     3.5
6.0     6.0     5.0
5.0     4.5     4.5
3       3.0     3.0
```

2.
```
sqrt(x + y)     pow(x, y + 7)       sqrt(area + fudge)
sqrt((time + tide) / nobody)        (-b + sqrt(b*b - 4*a*c))/(2*a)
abs(x - y)或者 labs(x - y)或者 fabs(x - y)
```

3.
```
// 计算 3.14159 的平方根
#include <iostream>
#include <cmath> // 提供 sqrt 和 PI
using namespace std;
int main()
{
    cout << "The square root of " << PI
         << " is " << sqrt(PI) << endl;
    return 0;
}
```

4. a.
```
// 判断编译器是否允许在#include 的#之前添加空格
    #include <iostream>
using namespace std;
int main()
{
    cout << "hello world" << endl;
    return 0;
}
```

b.
```
// 判断编译器是否允许在#和 include 之间插入空格
# include <iostream>
using namespace std;
// 其余代码和上例一样
```

5.
```
Wow
```

6. 函数声明：
```
int sum(int n1, int n2, int n3);
// 返回 n1，n2 和 n3 之和
```
函数定义：
```
int sum(int n1, int n2, int n3)
{
    return (n1 + n2 + n3);
}
```

7. 函数声明：
```
double ave(int n1, double n2);
// 返回 n1 和 n2 的平均值
```
函数定义：
```
double ave(int n1, double n2)
{
    return ((n1 + n2)/2.0);
}
```

8. 函数声明：
```
char positiveTest(double number);
// number 是正数，返回'P'
```

```
// number 是负数或者零，则返回'N'
```

函数定义：

```
char positiveTest(double number)
{
    if (number > 0)
        return 'P';
    else
        return 'N';
}
```

9. 假定函数定义了两个形参(param1 和 param2)。调用该函数时，我们向其传递相应的实参 arg1 和 arg2。这时，会在相应的形参位置代入 arg1 和 arg2 的值。也就是说，arg1 在 param1 的位置代入，arg2 在 param2 的位置代入。这样，函数中凡是使用形参的位置，都会替换成对应的实参。

10. 预定义(库)函数通常要求用#include 包含一个头文件。如果是自定义函数，程序员要么将函数代码放到和 main 相同的文件中；要么放到单独的文件，然后编译并链接到主程序。

11.
```
bool inOrder(int n1, int n2, int n3)
{
    return ((n1 <= n2) && (n2 <= n3));
}
```

12.
```
bool even(int n)
{
    return ((n % 2) == 0);
}
```

13.
```
bool isDigit(char ch)
{
    return ('0' <= ch) && (ch <= '9');
}
```

14.
```
bool isRootOf(int rootCandidate, int number)
{
    return (number == rootCandidate * rootCandidate);
}
```

15. 注释解释了函数的返回值，并给出了使用函数时程序员应该知道的其他信息。

16. 应用于函数定义时，过程抽象原则是指你写的函数应该能像一个黑盒那样使用。也就是说，使用该函数的程序员不需要查看函数定义主体来了解函数的工作原理。程序员要使用该函数，只需查看函数声明和注释。

17. 函数的用户将函数当作黑盒使用，意思是不需要查看函数定义的主体来了解它具体如何工作。使用函数只需查看函数声明和注释。

18. 为了增强对自己设计的程序的信心，应该使用已知正确答案的输入值测试它。正确答案可通过其他方式得到，比如使用纸笔或计算器。

19. 是等价的。无论哪种情况，函数都返回正确值，这两个定义完全符合黑盒设计要求(过程抽象原则)。

20. 要在函数定义中使用变量，应该在函数定义主体中声明它。

21. 答案都是肯定的。假定其他一切都正确，那么该程序能通过编译，能运行，运行时不会报错，能给出正确输出。

22. 函数能满足设计要求。具体地说，形参 inches 是传值参数。所以，就像本章正文讨论的那样，它是一个局部变量。因此，实参的值不会改变。

23. 函数声明：

```
double readFilter();
// 从键盘读入值，如果该值大于等于 0，就返回它；否则返回 0
```

函数定义：

```
// 使用 iostream
double readFilter()
```

```
{
    using namespace std;
    double valueRead;
    cout << "Enter a number:\n";
    cin >> valueRead;

    if (valueRead >= 0)
        return valueRead;
    else
        return 0.0;
}
```

24. 这个函数调用只有一个实参。所以它会使用只有一个参数的函数定义。
25. 这个函数调用有两个 double 类型的实参，所以它会使用函数声明中有两个 double 参数的函数(也就是第一个函数声明)。
26. 第二个实参是 int 类型，如有必要，第一个实参会被 C++自动转换为 double 类型，所以它将使用函数声明中第一个参数为 double 类型、第二个参数为 int 类型的函数(也就是第二个函数声明)。
27. 第二个实参是 double 类型，如有必要，第一个实参将由 C++自动转换为 double 类型，所以它将使用函数声明中有两个 double 参数的函数(也就是第一个函数声明)。
28. 不能这样做(至少不能以任何自然的方式)。表示正方形比萨和圆形比萨的最自然的方式是一样的；也就是说，都可以用一个数字来表示：对于圆形比萨来说是直径，对于正方形比萨来说是边长。无论哪种情况，unitPrice 函数都需要两个形参：一个 double 类型的形参表示价格；一个 int 类型的形参表示尺寸(直径或边长)。因此，两个函数声明的形参将具有相同的数量和类型(具体地说，两者都有一个 double 类型的形参和一个 int 类型的形参)。因此，编译器无法判断应该使用哪个定义。要挫败奸商的阴谋，可以定义两个函数，但两者要用不同的名称。
29. unitPrice 的定义不涉及任何输入或输出，所以没必要使用 iostream 库。程序 main 部分之所以需要 using 指令，是因为 cin 和 cout 在 iostream 中定义，而且那些定义将 cin 和 cout 放到了 std 命名空间中。

编 程 练 习

编程练习一般只需写很小的程序，运用本章提到的编程概念。

1. 1 公升等于 0.264179 加仑。写程序读入用户汽车消耗的汽油公升数和行驶的英里(哩)数，然后输出该车的哩/加仑指标。程序应该允许用户根据需要任意重复计算。定义函数来计算哩/加仑。程序应该为公升和加仑的换算关系使用全局定义常量。
2. 修改编程练习 1 的程序，获取两辆车的输入数据，输出每辆车的哩/加仑指标。程序应指出哪一辆车最省油(“哩/加仑”数最大)。
3. 美国股票价格有时以 1/8 美元为单位；例如，某只股票的价格可能是 $29\frac{7}{8}$ 或者 $89\frac{1}{2}$。写程序计算用户持有的一只股票的市值。程序要求输入该股票的持有量(以股为单位)，然后输入股价的整数部分和分数部分。输入分数部分时，要求用户输入两个 int 值：分别代表分子和分母。然后，程序输出用户持有的这只股票的市值。程序应该允许用户根据需要任意重复计算。要求在程序中包含一个函数定义，它接收 3 个 int 实参，分别是股价整数部分和构成分数部分的两个整数。函数返回代表每股单价的 double 值。
4. 写程序计算去年的通货膨胀率。该程序要求输入一件商品(比如一只热狗或者一克拉钻石)去年和今年的价格。两个价格的差值除以去年的价格，就是通货膨胀率。程序应该允许用户根据需要任意重复这个计算。定义函数来计算通货膨胀率。通货膨胀率应该是 double 值，该值代表百分数形式的通货膨胀率。例如，5.3 代表 5.3%。
5. 改进上个程序，打印商品在今后一年和两年内的估计价格。一年后，价格中上涨的部分等于通货膨胀

率乘以年初的价格。定义另一个函数来计算商品一年后的价格，参数是商品当前价格和通货膨胀率。

6. 为计算信用卡账户余额利息的函数写一个函数声明。函数获取的参数包括初始余额、月利率和计息月数。返回值是到期利息。不要忘了计算复利——也就是利滚利。每个月产生的利息都会加到余额中，下个月的利息将基于这个更大的余额进行计算。请使用一个与图 2.14 类似(但不必完全照搬)的 `while` 循环。将函数嵌入一个程序中。程序要求输入利率、初始账户余额和月数，然后输出到期利息。程序允许任意重复计算，直到用户要求终止程序。

7. 视频讲解：*Solution to Practice Program 4.7*

   两个物体距离为 d，质量分别为 m_1 和 m_2，它们之间的万有引力为：

$$F = \frac{Gm_1m_2}{d^2}$$

   其中 G 为万有引力常量：

$$G = 6.673 \times 10^{-8} \mathrm{cm}^3/(\mathrm{g.sec}^2)$$

   写一个函数定义获取两个物体的质量和距离，返回两者之间的万有引力。既然要用到上述公式，那么应该知道万有引力的单位是达因。1 达因[①]等于 $1\mathrm{g.cm/sec}^2$ 。

   应该为万有引力常量使用全局定义的常量。将函数定义嵌入一个完整的程序中。该程序基于给定的输入来计算两个物体之间的万有引力。程序应该允许用户根据需要任意重复计算。

8. 之所以有好几个绝对值函数，完全是历史的失误。C 库在 C++问世时就已经有了。由于用起来很顺手，C++没有通过函数重载来重写它们。请找出所有绝对值函数，通过重载函数名 `abs` 将它们全部重写一遍。最起码，函数能对 `int`、`long`、`float` 和 `double` 类型的值求绝对值。

9. 写重载的 `max` 函数获取 `double` 类型的两个或三个参数，返回其中最大的数。

编 程 项 目

编程项目要求综合运用多方面的知识来解决问题，程序一般比编程练习大，解题方式多样化。

1. 写程序计算一幢新房子第一年的税后支出。房主的支出等于年抵押还款(annual mortgage cost)扣除减免税额(tax savings)。输入的是房屋价格(price)和首付款(down payment)。年抵押还款为初始贷款余额的 3%(用于偿还贷款本金)，再加上初始贷款余额的 8%(用于支付利息)。初始贷款余额(initial loan balance)等于房价减去首付款。假定为你的个人所得征收的边际税率(marginal tax)为 35%，并假定贷款利息支出可用于抵税[②]。所以，减免税额就是利息支出的 35%。程序应该使用至少两个函数定义。应该允许根据需要任意重复计算。

2. 写程序要求用户输入身高、体重和年龄，根据以下公式计算该用户适合穿什么尺码的服装。

   - 帽子的尺码 = 体重(以磅计)除以高度(以英寸计)，结果乘以 2.9。
   - 上衣尺码(以英寸计的胸围尺寸) = 身高乘以体重除以 288(年龄在 30 岁以上的人士要进行修正，即每 10 年多加 1/8 英寸)。注意，只有在满了 10 年之后才进行这个修正(例如，年龄在 30～39 岁之间，不需要任何修正；但年龄到了 40 岁，就要增加 1/8 英寸)。
   - 腰围尺码 = 体重除以 5.7(年龄在 28 岁以上的人士要进行修正，即每 2 年增加 1/10 英寸)。注意，只有在满了 2 年之后才进行这个修正(例如，29 岁不需要修正，30 岁就要增加 1/10 英寸)。

   为每个计算都使用一个函数。程序允许任意重复计算。

3. 修改编程项目 2 的程序，使它能计算用户 10 年后的上衣和腰围尺码。

---

① 高中物理讲到，1 达因的力相当于促使质量为 1 克的物体获得 1 厘米/秒 ^{2} 的加速度的力。——译注

② 美国政府规定，征收个人所得税时，房屋抵押贷款的利息支出可用于抵税。——译注

4. 写程序输出“Ninety-Nine Bottles of Beer on the Wall”这首歌的歌词。程序应该以英语单词(而非阿拉伯数字)的形式打印酒瓶数。例如:

```
Ninety-nine bottles of beer on the wall,
Ninety-nine bottles of beer!
Take one down, Pass it around,
Ninety-eight bottles of beer on the wall.
…
One bottle of beer on the wall,
One bottle of beer!
Take one down, Pass it around,
Zero bottles of beer on the wall.
```

用一个函数来设计程序，它获取 0～99 的整数，返回一个字符串，其中包含该整数值的英语单词形式。不要给函数写 100 个不同的 if-else 语句！相反，用%和/提取十位和个位来构造英语单词字符串。一些值需特殊处理，比如 0，10～19，等等。

5. 为保持体重，成年人每天要消耗足够的卡路里，从而(1)满足基础代谢率(呼吸、维持体温等所需的能量)，(2)提供体力活动(比如身体锻炼)所需能量，以及(3)提供消化食物所需能量。对于体重 *P* 磅的成年人，可用以下公式估算每天的卡路里需求：

(1) 基础代谢率：所需卡路里 $= 70 * (P/2.2)^{0.756}$

(2) 体力活动：所需卡路里 = 0.0385 * *剧烈度* * *P* * *分钟*
其中，*分钟*是花在体力活动上的分钟数，*剧烈度*是估算体力活动剧烈程度的数值。以下是一些示例值：

活动	剧烈度
跑步 10 mph	17
跑步 6 mph	10
篮球	8
散步 1mph	1

(3) 消化食物：所需的卡路里 = *消耗的总卡路里* * 0.1

换言之，我们消耗的卡路里 10%用在消化食物上。

写函数计算基础代谢率所需的卡路里，它获取人的体重作为参数。写另一个函数计算体力活动所需的卡路里，获取的参数包括剧烈度、体重和花在体力活动上的分钟数。

在程序中输入一个人的体重、估计的体力活动剧烈度、花在体力活动上的分钟数以及一份你喜爱的食物所含的卡路里数。随后，程序应计算并输出在当前的体力活动等级上，每天应该吃多少份那种食物，才能维持一个人的当前体重。计算时，应考虑到消化食物所需的能量。

网上能找到许多食物的卡路里数。例如，一个双层芝士汉堡的卡路里数约为 1000。

6. 你发明了一台自动贩卖机向顾客出售油炸脆皮鲜奶。写程序模拟这台贩卖机。一块油炸脆皮鲜奶的价格是 3.5 美元，机器只接受以下面值的硬币：1 dollar(1 美元)，1 quarter(25 美分)，1 dime(1 角)或者 1 nickel(5 美分)。程序应模拟一个人不断将硬币投入贩卖机的过程，反复提示用户投入下一枚硬币。每投入一枚硬币，都提示总共已投了多少钱。一旦投了 3.5 美元或者更多，就输出以下提示：“请享用你的油炸脆皮鲜奶”，输出应该找多少零钱。使用“自顶向下设计法”确定程序需要哪些函数。

7. 你发明了一台时间机器，能回到最多 24 小时之前。具体回到什么时间以分钟为单位指定。为了向机器输入正确分钟数，程序需获取一个起始时间(使用小时数、分钟数和一个代表早上或下午的布尔值)和一个未来时间(使用小时数、分钟数和一个代表早上或下午的布尔值)，并计算起始和未来时间差值。

具体地说，程序中的一个时间要用 3 个变量指定：

```
int hours, minutes;
bool isAM;
```

例如，为了表示 11:50 PM，应存储以下这些值：

```
hours = 11;
minutes = 50;
isAM = false;
```

这意味着总共需要 6 个变量存储起始时间和未来时间。

写程序允许用户输入起始和未来时间。写一个名为 computeDifference 的函数来获取 6 个变量作为参数。函数以 int 的形式返回两个时间相差多少分钟。

例如，假定起始时间是 11:59 AM，未来时间是 12:01PM，程序计算出时间差值是 2 分钟。起始时间是 11:59 AM，未来时间是 11:58 AM，程序则应计算出时间差值是 1439 分钟(23 小时 59 分钟)。

用户应输入 AM 表示上午，输入 PM 表示下午。程序应读取两个字符值来识别 AM 或 PM(图 2.3 演示了字符输入)。字符可以像数字那样进行比较。例如，假定变量 aChar 是 char 类型，那么一旦 aChar 包含字母 A，(aChar == 'A')这个布尔表达式就会求值为 true。

8. 视频讲解：*Solution to Programming Project 4.8*

   修改第 3 章的编程项目 11，使用名为 containsDigit 的函数判断数字中是否包含特定数位。函数声明如下所示：

```
bool containsDigit(int number, int digit);
```

   如 number 中包含 digit，函数返回 true；否则返回 false。程序利用这个函数查找可通过烤箱上的按键输入的最接近的数字。

9. 联盟使用以下乐透系统让表现最差的 4 个队选秀：
   - 最差的队得 20 个球
   - 倒数第二差的队得 10 个球
   - 倒数第三差的队得 6 个球
   - 倒数第四差的队得 4 个球

   随机选一个球来确定第一签(称为状元签)。拥有该球的队获得最优先的选秀权。将球放回去。第二签还是随机选球。如果球属于刚才抽中第一签的队，就把球放回去。重复这个过程直到球不属于抽中第一签的队。

   如此重复，直到确定全部 4 个队的选秀顺序。

   写函数获取当前要进行选秀排位的队，根据乐透规则模拟选一个球，返回该球属于哪个队。自己决定如何设计函数来执行这些操作。在 main 函数中输出最终的选秀顺序，例如：

```
second to-last picks 1, last place picks 2, third-to-last picks 3, and fourth-to-last picks 4
```

   或者：

   倒数第二队第 1 签，最后一队第 2 签，倒数第三队第 3 签，倒数第四队第 4 签

   如果每次运行都改变随机种子值，那么每次显示的顺序都应该不同。

   一个稍难的版本是允许用户输入 4 个队的名字。程序输出选秀顺序时应使用这些名字。

10. 重做第 3 章的编程项目 14，改为写函数获取糖、Edoc 和 Margorp 的数量，返回能获得的经验值。修改程序向玩家提示使用以下规则：
    - 多抓一只 Edoc 得 3 块 Edoc 糖。
    - 基于输入如果无法获得任何经验值就输出：“任何 Edoc 都进化不了，去抓更多吧。”
    - 和现在进化相比，多抓一只或两只 Edoc 能提供更多经验值，就输出：“先不要进化，抓更多 Edoc。”
    - 否则输出：“该进化了”。

第 5 章　用函数完成所有子任务

一切皆有可能。

——*流行语*

概述

第 4 章讨论了自顶向下设计策略，它是为程序设计算法的有效方式。可将程序的任务分解成子任务，再将这些子任务的算法作为函数实现。前面解释了如何定义函数，在函数调用中传递实参，让函数返回一个值。计算单个值的子任务是非常重要的子任务，但并非唯一的一种。本章要完成对 C++函数的描述，展示如何设计执行其他子任务的函数。

预备知识

学习本章之前，应该完成第 2～4 章的学习。

5.1　void 函数

子任务在 C++中作为函数实现。第 4 章讨论的函数总是返回一个值，但还有其他形式的子任务。有的可能生成多个值，有的则不生成任何值。C++要求函数要么返回一个值，要么什么值都不返回。如本章后文所述，需要生成多个值的子任务通常要实现成不返回任何值的函数(听起来自相矛盾)。目前为了避免这种复杂性，先将重点放在不生成任何值的子任务上，并解释具体如何实现。不返回值的函数称为 void 函数。例如，程序的一个典型的子任务是输出计算结果。直接在屏幕上输出结果，不生成供程序其他部分使用的值。这种子任务适合用 void 函数实现。

void 函数的定义

C++ void 函数的定义方式类似于返回值的函数。例如下面这个 void 函数：

```
void showResults(double fDegrees, double cDegrees)
{
    using namespace std;
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(1);
    cout << fDegrees
         << " degrees Fahrenheit is equivalent to\n"
         << cDegrees << " degrees Celsius.\n";
    return;
}
```

函数输出华氏温度换算为摄氏温度的结果，但它本身不负责实际换算。实际换算在程序其他地方进行。该 void 函数只实现了“输出换算结果”子任务。目前暂不关心实际如何换算。

如上述函数定义所示，void 函数定义与第 4 章讨论的函数定义相比，只有两个区别。其一，原来指定返回值类型的地方使用了关键字 void。这向编译器表明该函数不返回任何

值。void(空)一词可理解成：“该函数不返回值”。其二，return 后面没有添加代表返回值的表达式，因为 void 函数本来就不返回任何值。void 函数的语法总结请参见图 5.1。

图 5.1　void 函数定义的语法

void 函数声明

```
void Function_Name(Parameter_List);
```

void 函数定义

```
    void Function_Name(Parameter_List)  <-------- 函数头
    {
        Declaration_1
        Declaration_2    <-------- 声明和可执行语句可以混合在一起
        . . .
主体     Declaration_Last
        Executable_Statement_1      <--
        Executable_Statement_2         |-- 可以包含(也可以不包含)一个
        . . .                          |   或多个 return 语句
        Executable_Statement_Last   <--
    }
```

void 函数调用是可执行语句。例如，showResults 函数可以这样调用：

```
showResults(32.5, 0.3);
```

在程序中执行以上语句，会导致屏幕上显示以下输出：

```
32.5 degrees Fahrenheit is equivalent to
0.3 degrees Celsius.
```

注意，函数调用以分号结尾，告诉编译器函数调用是可执行语句。

调用 void 函数时，实参(参数值)会替换形参，然后执行函数主体内的语句。例如，调用 void 函数 showResults 时，会导致一些输出被写到屏幕。调用 void 函数，就好比将函数定义主体复制到程序，并取代原来的函数调用。调用这个函数时，实参会替换形参。然后，可假定函数主体已成为程序的一部分。

无参函数完全合法，有时还相当有用。这种函数在声明中不列出任何形参。调用时也不传递实参。例如以下 void 函数 initializeScreen，它只是向屏幕发送一个换行指令：

```
void initializeScreen()
{
    using namespace std;
    cout << endl;
    return;
}
```

如果某程序的第一个可执行语句就是以下函数调用：

```
initializeScreen();
```

上一个程序的输出就肯定会和当前程序的输出分开[①]。

必须注意，即使函数没有参数，也必须在函数声明和函数调用中包括圆括号。下一节的编程实例将在一个完整的程序中展示这两个示例 void 函数。

① 在某些系统中，如果上一个程序最后没有输出换行符，下个程序的输出可能和它混在一起。详情参见 2.2.4 节。——译注

编程实例　温度换算

图 5.2 的程序取华氏温度作为输入，输出等价摄氏温度。华氏温度 F 与摄氏温度 C 的换算公式如下：

$$C = (5/9)(F - 32)$$

图 5.2 的 celsius 函数使用该公式进行温度换算。

图 5.2　void 函数

```
1  // 这个程序将华氏温度换算为摄氏温度
2  #include <iostream>
3
4  void initializeScreen();
5  // 将当前输出与上一个程序的输出分开
6
7
8  double celsius(double fahrenheit);
9  // 将华氏温度换算为摄氏温度
10
11
12 void showResults(double fDegrees, double cDegrees);
13 // 显示输出，假定摄氏温度 cDegrees 等价于华氏温度 fDegrees
14
15
16 int main()
17 {
18     using namespace std;
19     double fTemperature, cTemperature;
20
21     initializeScreen();
22     cout << "I will convert a Fahrenheit temperature"
23          << " to Celsius.\n"
24          << "Enter a temperature in Fahrenheit: ";
25     cin >> fTemperature;
26
27     cTemperature = celsius(fTemperature);
28
29     showResults(fTemperature, cTemperature);
30     return 0;
31 }
32
33 // 函数定义使用了 iostream:
34 void initializeScreen()
35 {
36     using namespace std;
37     cout << endl;
38     return;  <-------------------- 这个 return 语句可选
39 }
40 double celsius(double fahrenheit)
41 {
42     return ((5.0/9.0)*(fahrenheit - 32));
43 }
44 // 函数定义使用了 iostream:
45 void showResults(double fDegrees, double cDegrees)
46 {
47     using namespace std;
48     cout.setf(ios::fixed);
49     cout.setf(ios::showpoint);
50     cout.precision(1);
51     cout << fDegrees
52          << " degrees Fahrenheit is equivalent to\n"
53          << cDegrees << " degrees Celsius.\n";
54     return;  <-------------------- 这个 return 语句可选
55 }
```

示范对话

```
I will convert a Fahrenheit temperature to Celsius.
Enter a temperature in Fahrenheit: 32.5
32.5 degrees Fahrenheit is equivalent to
0.3 degrees Celsius.
```

void 函数中的 return 语句

void 函数和返回值的函数都可包含 return 语句。如函数有返回值，就用 return 语句指定。void 函数的 return 语句只是终止函数调用。如前一章所述，有返回值的函数要执行 return 语句来终止。但 void 函数可以不包含 return 语句。如果不包含，函数体执行完后将自动终止——可认为在函数体的结束花括号之前有一个隐式的 return 语句。例如，图 5.2 的 initializeScreen 和 showResults 函数完全可以在函数定义中省略 return 语句，执行结果完全一样。

虽然函数主体的结束花括号之前有一个隐式的 return 语句，但并不是说永远用不着在 void 函数中包含 return 语句。例如，图 5.3 的函数定义可作为一个酒店管理程序的一部分使用。针对一桌客人，函数输出如何对给定重量的冰淇淋进行分配。如餐桌旁没人(即 number 等于 0)，if 语句内的 return 语句就终止函数调用，避免出现除以 0 的情况。如 number 不等于 0，就在执行函数体的最后一个 cout 语句后结束函数调用。

图 5.3　在 void 函数中使用 return 语句

函数声明

```
1  void iceCreamDivision(int number, double totalWeight);
2  // 针对为 number 位客人分配总重量为 totalWeight 盎司的冰淇淋，输出相应的指示
3  // 如 number 等于 0，就什么事情都不做
```

函数定义

```
1  // 函数定义使用了 iostream:
2  void iceCreamDivision(int number, double totalWeight)
3  {
4      using namespace std;
5      double portion;
6
7      if (number == 0)
8          return;              <——— 如 number 等于 0，函数执行就在此处终止
9      portion = totalWeight/number;
10     cout.setf(ios::fixed);
11     cout.setf(ios::showpoint);
12     cout.precision(2);
13     cout << "Each one receives "
14          << portion << " ounces of ice cream." << endl;
15 }
```

你现在可能已猜出程序 main 部分其实就是一个名为 main 的函数的定义。运行程序会自动调用 main 函数，它继续调用其他函数。技术上说，main 是返回 int 值的函数，所以需要 return 语句。但使用 main 函数感觉像是使用一个 void 函数，似乎用不到它的返回值。将程序 main 部分视为返回整数的函数也许令人不可思议，但那是传统！最好还是将程序 main 部分看成是“程序主干”，不要纠缠于这些细节。[①]

① C++标准规定，main 函数可以省略返回语句，等价于返回 0，但许多(老的)编译器仍然需要该语句。

√ 自测题

1. 以下程序的输出是什么？

```
#include <iostream>
void friendly();
void shy(int audienceCount);
int main()
{
    using namespace std;
    friendly();
    shy(6);
    cout << "One more time:\n";
    shy(2);
    friendly();
    cout << "End of program.\n";
    return 0;
}

void friendly()
{
    using namespace std;
    cout << "Hello\n";
}

void shy(int audienceCount)
{
    using namespace std;
    if (audienceCount < 5)
        return;
    cout << "Goodbye\n";
}
```

2. void 函数定义是否必须包含 return 语句？
3. 对于图 5.2 中的 initializeScreen 函数，在定义中省略 return 语句对程序有什么影响？程序能通过编译吗？能正常运行吗？程序行为是否有所不同？省略 showResults 函数定义中的 return 语句，结果又如何？省略 celsius 函数定义中的 return 语句呢？
4. 写一个 void 函数的函数定义，该函数获取 3 个 int 类型的实参，并在屏幕中输出它们的乘积。把函数定义放到一个完整程序中。程序读取 3 个数字，再调用该函数。
5. 你所用的编译器是否同时允许 void main()和 int main()？如使用 int main()，但不提供 return 0;语句，编译器会发出什么警告？要得出结论，请写几个小测试程序，可能需要咨询老师或者身边其他懂行的人。
6. void 函数调用是作为语句还是表达式使用？

5.2　传引用参数

调用函数时，实参会替换函数定义中的形参。通俗地说，实参在形参位置“代入”。有几种代入机制。第 4 章和到本章目前为止使用的都是传值调用(call-by-value)。第二种机制是传引用调用(call-by- reference)。

初探传引用调用

以前使用的传值调用机制不适合某些子任务。例如，一个常见子任务是从用户处获取一个或多个输入值。回顾图 5.2 的程序，它的任务被分解为 4 个子任务：初始化屏幕，获

取华氏温度，计算对应摄氏温度，以及输出结果。4 个子任务中，3 个分别由 initializeScreen，celsius 和 showResults 函数实现。但“获取输入”子任务通过以下 4 行代码实现(而不是通过函数调用)：

```
cout << "I will convert a Fahrenheit temperature"
     << " to Celsius.\n"
     << "Enter a temperature in Fahrenheit: ";
cin >> fTemperature;
```

“获取输入”子任务也应通过函数调用来进行，这就需要使用传引用参数。

获取用户输入的函数应该将一个或多个变量设为输入值，所以函数调用应该将一个或多个变量作为实参，而且应该能更改它们的值。如果还是使用传值形参，函数调用中的实参确实可以是变量，但函数只是获取该变量的值，无法以任何方式更改变量。也就是说，使用传值形参，代入形参的只是实参的值(而非实参变量本身)。对于输入函数，我们希望用变量本身(而非变量值)替换形参。传引用调用机制满足了这一点。使用**传引用调用形参**(简称**传引用参数**)，函数调用中相应的实参必须是变量，代入形参的是该变量。这好比将实参变量原封不动地复制到函数定义中并取代形参。此时执行函数主体代码，就可更改实参变量的值了。

必须以某种方式标记传引用参数，使编译器将其与传值参数区分开。为了标记传引用参数，在函数声明和函数定义头中，要在类型名称末尾附加符号&。例如，以下函数定义的形参 fVariable 就是传引用参数：

```
void getInput(double& fVariable)
{
    using namespace std;
    cout << "I will convert a Fahrenheit temperature"
         << " to Celsius.\n"
         << "Enter a temperature in Fahrenheit: ";
    cin >> fVariable;
}
```

在含有这个函数定义的程序中，下面的函数调用将 fTemperature 变量设为从键盘读入的值：

```
getInput(fTemperature);
```

可利用该函数定义修改图 5.2 的程序来完成“获取输入”子任务。但下面不是修改旧程序，而是写了一个全新的程序。

图 5.4 演示了传引用参数。程序做的事情不多，就是读入两个数字，以相反顺序输出。getNumbers 函数和 swapValues 函数的参数是传引用参数。输入由以下函数调用执行：

```
getNumbers(firstNum, secondNum);
```

该函数设置变量 firstNum 和 secondNum 的值。之后，以下函数调用交换变量 firstNum 和 secondNum 的值：

```
swapValues(firstNum, secondNum);
```

后续几个小节将详细描述传引用调用机制，并解释图 5.4 所用的函数。

图 5.4　传引用参数

```
1  // 演示传引用参数的程序
2  #include <iostream>
3  void getNumbers(int& input1, int& input2);
4  // 从键盘读取两个整数
5  void swapValues(int& variable1, int& variable2);
6  // 交换 variable1 和 variable2 的值
7  void showResults(int output1, int output2);
8  // 依次显示 variable1 和 variable2 的值
9  int main()
10 {
11     int firstNum, secondNum;
12
13     getNumbers(firstNum, secondNum);
14     swapValues(firstNum, secondNum);
15     showResults(firstNum, secondNum);
16     return 0;
17 }
18 // 使用 iostream:
19 void getNumbers(int& input1, int& input2)
20 {
21     using namespace std;
22     cout << "Enter two integers: ";
23     cin >> input1
24         >> input2;
25 }
26 void swapValues(int& variable1, int& variable2)
27 {
28     int temp;
29     temp = variable1;
30     variable1 = variable2;
31     variable2 = temp;
32 }
33 // 使用 iostream:
34 void showResults(int output1, int output2)
35 {
36     using namespace std;
37     cout << "In reverse order the numbers are: "
38          << output1 << " " << output2 << endl;
39 }
```

示范对话

```
Enter two integers: 5 10
In reverse order the numbers are: 10 5
```

传引用调用详解

大多数时候都可以简单地认为，在传引用调用机制中，作为实参提供的变量名直接替换了传引用形参。但实际过程要复杂一些，而且有时必须意识到这一点。所以，现在更深入地探讨一下传引用调用的参数替换过程。

以前说过，程序变量作为内存位置实现。编译器为每个变量都分配一个内存位置。例如，图 5.4 的程序在编译时，可能将 1010 这个位置分配给 firstNum 变量，将 1012 这个位置分配给 secondNum 变量。出于举例的目的，假定变量真的存储在这些位置。也就是说，执行以下代码：

```
int firstNum = 0, secondNum = 0;
```

内存位置 1010 和 1012 将存储 0 值。下图的箭头指出变量引用的内存位置。

内存位置	值	
. . .		
1008		
1010	0	← *firstNum*
1012	0	← *secondNum*
1014		
. . .		

接着考虑图 5.4 的以下函数声明：

```
void getNumbers(int& input1, int& input2);
```

传引用参数 input1 和 input2 是函数调用的实参占位符。

传引用调用

要使形参成为**传引用**参数，需在其类型名称后添加符号&。对应地，在函数调用中，传递的实参应该是变量，而不能是常量或其他表达式。调用函数时，实参变量(而不是该变量的值)会替换形参。调用函数期间，函数主体对形参的任何更改都会作用于实参变量。本章详细讨论了这种替换机制。

示例(函数声明中使用了传引用参数)

```
void getData(int& firstIn, double& secondIn);
```

现在分析图 5.4 的以下函数调用：

```
getNumbers(firstNum, secondNum);
```

执行该函数调用，函数得到的不是 firstNum 和 secondNum 中存储的值。相反，得到的是和每个名称关联的内存位置。在本例中，位置列表如下：

```
1010
1012
```

这是分配给实参变量 firstNum 和 secondNum 的内存位置(注意顺序)。和形参关联的实际是这些内存位置。第一个内存位置与第一个形参关联，第二个与第二个关联，依此类推。本例中，input1 是第一个参数，得到和 firstNum 相同的内存位置。input2 是第二个参数，得到和 secondNum 相同的内存位置，如下图所示。

	内存位置	值	
	. . .		
	1008		
input1 →	1010	0	← *firstNum*
input2 →	1012	0	← *secondNum*
	1014		
	. . .		

执行函数主体语句时，函数主体对形参执行的任何操作实际会作用于与形参关联的内存位置处的变量。本例中，getNumbers 函数主体的指令要求用 cin 语句将一个值存储到形参 input1 中，那个值实际会存储到内存位置 1010 的变量中(已知这个变量是 firstNum)。类似地，getNumbers 函数主体的指令还要求用 cin 语句将一个值存储到形参 input2 中，那个值实际会存储到内存位置 1012 的变量中(已知这个变量是 secondNum)。所以，函数指示计算机对 input1 和 input2 所做的一切实际会作用于 firstNum 和 secondNum 变量。例如，执行图 5.4 的程序并输入 5 和 10，结果如下图所示。

	内存位置	值	
	...		
	1008		
input1 →	1010	5	← *firstNum*
input2 →	1012	10	← *secondNum*
	1014		
	...		

getNumbers 函数退出时，变量 input1 和 input2 因离开作用域而被丢弃。这意味着不能继续通过 input1 和 input2 获取 1010 和 1012 处的数据值。但数据仍然存在于内存位置 1010 和 1012，可在 main 函数的作用域中通过 firstNum 和 secondNum 变量来访问。图 5.5 展示了传引用调用机制在 getNumbers 函数调用中的详细工作原理。

图 5.5　传引用实参的行为

解析传引用的函数调用(参考图 5.4)

0. 假定编译器已为 firstNum 和 secondNum 变量分配了以下内存地址：

```
firstNum → 1010
secondNum → 1012
```

(我们不知道实际分配的地址，而且结果与实际地址无关。但通过杜撰两个地址，有助于更形象地思考这个问题，也使你更容易理解。)

1. 在图 5.4 的程序中，开始执行以下函数调用：

```
getNumbers(firstNum, secondNum);
```

2. 要求函数使用 firstNum 变量的内存位置取代形参 input1，用 secondNum 变量的内存位置取代形参 input2。这相当于函数定义被重写为以下代码(不是合法的 C++代码，但更容易理解)：

```
void getNumbers(int& <内存位置 1010 处的变量>, int& <内存位置 1012 处的变量>)
{
    using namespace std;
    cout << "Enter two integers: ";
    cin >> <内存位置 1010 处的变量>
        >> <内存位置 1012 处的变量>;
}
```

由于内存位置 1010 和 1012 保存的变量分别是 firstNum 和 secondNum。所以，函数定义相当于重写为以下形式：

```
void getNumbers(int& firstNum, int& secondNum)
{
    using namespace std;
    cout << "Enter two integers: ";
    cin >> firstNum
        >> secondNum;
}
```

3. 执行函数主体。效果相当于执行以下代码：

```
{
    using namespace std;
    cout << "Enter two integers: ";
    cin >> firstNum
        >> secondNum;
}
```

4. 执行 cin 语句时，firstNum 和 secondNum 变量的值被设为从键盘输入的值(如对话如图 5.4 所示，则 firstNum 的值被设为 5，secondNum 的值被设为 10)。

5. 函数调用终止时，firstNum 和 secondNum 变量保持了它们在函数主体中由 cin 给定的值(如对话如图 5.4 所示，那么在函数调用终止时，firstNum 的值是 5，secondNum 的值是 10)。

还有一些细节需要说明。既然 firstNum 就是内存位置为 1010 的变量，为什么还要坚持说“内存位置 1010 处的变量”，而不是直接称之为“firstNum”？原因是，实参和形参名称相近可能引起混淆，使用内存位置就有助于澄清。例如，getNumbers 函数的形参是 input1 和 input2。假设修改图 5.4 的程序，让 getNumbers 函数接收两个同样名为 input1 和 input2 的实参。另外，假设函数中采取的操作无法让人一目了然。例如，假设把输入的第一个数保存到名为 input2 的变量中，把第二个数保存到名为 input1 的变量(可能因为第二个数先处理，也可能因为第二个数更重要)。假定 input1 和 input2 变量已在程序的 main 部分声明，而且分配到了 1014 和 1016 这两个内存地址。函数调用如下：

```
int input1, input2;
getNumbers(input2, input1);        ←——————— 注意参数顺序
```

此时如果说“input1”，就不知道指的是程序 main 部分声明的变量 input1，还是形参 input1。但是，如果 main 部分声明的 input1 变量已分配内存地址 1014，那么“内存位置 1014 处的变量”就十分明确，不会产生任何歧义。下面仔细分析一下这种情况下的参数替换机制。

在这个调用中，对应于形参 input1 的实参是变量 input2，对应于形参 input2 的实参是变量 input1。这把人搞糊涂了，但对计算机而言根本不是问题，因为计算机从来不会真正地“用 input2 替换 input1”或者“用 input1 替换 input2”。计算机只操纵内存位置。计算机用“内存位置 1016 处的变量”替换形参 input1，用“内存位置 1014 处的变量”替换形参 input2。

编程实例　swapValues 函数

图 5.4 定义的 swapValues 函数交换两个变量中存储的值。我们使用以下函数声明和注释描述该函数：

```
void swapValues(int& variable1, int& variable2);
// 交换 variable1 和 variable2 的值
```

为理解函数的工作原理，假定 firstNum 变量的值是 5，而 secondNum 变量的值是 10，然后分析以下调用：

```
swapValues(firstNum, secondNum);
```

完成这个函数调用之后，firstNum 的值变成 10，secondNum 的值变成 5。

如图 5.4 所示，swapValues 函数的定义使用了名为 temp 的局部变量。该局部变量必不可少。有人想当然地将函数定义简化为以下形式：

```
void swapValues(int& variable1, int& variable2)
{
   variable1 = variable2;
   variable2 = variable1;      这没有用！
}
```

要弄明白为什么没有用，请思考使用上述定义后，执行以下函数调用会发生什么：

```
swapValues(firstNum, secondNum);
```

firstNum 和 secondNum 变量会替换形参 variable1 和 variable2。所以，如果使用这个不正确的函数定义，函数调用等价于：

```
firstNum = secondNum;
secondNum = firstNum;
```

这些代码不能产生我们希望的结果。firstNum 的值被设为 secondNum 的值，这是正确的。随即，secondNum 的值被设为修改后的 firstNum 的值，而 firstNum 现在存储的就是 secondNum 的原始值。所以，secondNum 的值根本没变(如果还是想不通，试一试为 firstNum 和 secondNum 变量代入具体的值)。函数为了完成它既定的任务，必须保存 firstNum 的原始值，使其不会被丢失。这正是在函数的正确定义中使用局部变量 temp 的目的(参见图 5.4)。使用正确的函数定义时，如果用实参 firstNum 和 secondNum 调用函数，则函数调用等价于以下代码，它能正确完成任务：

```
temp = firstNum;
firstNum = secondNum;
secondNum = temp;
```

形参与实参

任何与形参和实参有关的术语都可能令人迷惑不解。但如果牢记以下三点，就能轻松加以区分。

其一，函数**形参**在函数声明中列出，在函数定义主体中使用。任何形参都是一种空白位置或者占位符，会在函数调用时被别的东西填充。

其二，**实参**用于填充形参。写函数调用时，实参在函数名后的圆括号中列出。执行函数调用时，实参会在形参的位置“代入”(或者说“插入”)。

其三，“传值调用”和“传引用调用”这两个术语是“代入”过程的具体机制。**传值调用**机制只使用实参的值。这种机制中，向形参(局部变量)传递的是实参的值。而**传引用调用**机制传递的是变量本身。这种机制中，实参变量会替换形参，故对形参的任何更改实际都会作用于实参变量。

混合的参数列表

形参到底是传值还是传引用，取决于它的类型名称后是否有符号&。有就是传引用参数，没有则是传值参数。

在同一个函数中混合使用传值和传引用参数完全合法。例如在以下函数声明中，第一个和最后一个参数就是传引用参数，中间的是传值参数：

```
void goodStuff(int& par1, int par2, double& par3);
```

传引用参数并非只能在 void 函数中使用，在有返回值的函数中也能使用。所以，如果函数使用了传引用参数，就既能更改实参变量的值，也能返回一个值。

编程提示：应该使用哪种参数

视频讲解：*Call by Reference and Call by Value*

图 5.6 展示了编译器如何区别对待传值和传引用参数。函数体中，par1Value 和 par2Ref 参数被分别赋了一个值。由于是两种不同的参数，所以结果也不同。

par1Value 是传值参数，所以是局部变量。像下面这样调用函数：

```
doStuff(n1, n2);
```

局部变量 par1Value 被初始化为 n1 的值(1)。然后，函数会忽略变量 n1。从示范对话看出，形参 par1Value 在函数体中被设为 111，然后在屏幕上输出该值。但实参 n1 的值没有改变。如示范对话所示，n1 仍然是原来的值 1。

另一方面，par2Ref 是传引用参数。调用函数时，实参变量 n2 本身(而非它的值)会替换形参 par2Ref。所以，执行以下代码：

```
par2Ref = 222;
```

等同于执行：

```
n2 = 222;
```

所以，执行函数体时，变量 n2 的值被改变了。如示范对话所示，函数调用将 n2 的值从 2 变成 222。

琢磨一下图 5.6 的程序，很容易判断具体应该使用哪一种参数传递机制。如果希望函数更改变量的值，与那个变量对应的形参必须是传引用参数，而且类型必须用符号&标记。其他所有情况都可使用传值参数。 ■

图 5.6　比较两种参数传递机制

```
1  // 展示传值和传引用参数的区别
2
3  #include <iostream>
4  void doStuff(int par1Value, int& par2Ref);
5  // par1Value 是传值参数，par2Ref 是传引用参数
6
7  int main()
8  {
9      using namespace std;
10     int n1, n2;
11
12     n1 = 1;
13     n2 = 2;
14     doStuff(n1, n2);
15     cout << "n1 after function call = " << n1 << endl;
16     cout << "n2 after function call = " << n2 << endl;
17     return 0;
```

```
18  }
19  void doStuff(int par1Value, int& par2Ref)
20  {
21      using namespace std;
22      par1Value = 111;
23      cout << "par1Value in function call = "
24           << par1Value << endl;
25      par2Ref = 222;
26      cout << "par2Ref in function call = "
27           << par2Ref << endl;
28  }
```

示范对话

```
par1Value in function call = 111
par2Ref in function call = 222
n1 after function call = 1
n2 after function call = 222
```

陷阱：疏忽造成的局部变量

假如希望函数更改变量的值，则相应的形参必须是传引用参数，而且必须在类型名称后附加&。如果不小心遗漏了&，它实际就是传值参数。程序运行时，会发现函数调用并没有更改对应实参的值。这是由于传值参数是局部变量，所以在函数调用中更改它的值时，和其他所有局部变量一样，更改只在函数主体内有效，不会在主体外部反映出来。由于遗漏符号&往往是因疏忽而致，所以这是一个难以发现的逻辑错误。程序“看起来没错”，但已经因为疏忽造成了一个本来不该存在的局部变量。

例如，图 5.7 的程序和图 5.4 的程序基本相同，只是 swapValues 函数遗漏了符号&。所以，variable1 和 variable2 是局部变量。firstNum 和 secondNum 这两个实参变量本身永远不会替换 variable1 和 variable2。相反，用于替换 variable1 和 variable2 的是 firstNum 和 secondNum 的值。随后，variable1 和 variable2 的值进行交换。然而，firstNum 和 secondNum 的值还是原封未动。省略两个符号&，就致使程序产生了根本性的错误，即使它看起来和正确的程序几乎一模一样，而且还能正常编译和运行，不报告任何错误消息。■

图 5.7　疏忽所造成的局部变量

```
 1  // 演示传引用参数的程序
 2  #include <iostream>
 3  void getNumbers(int& input1, int& input2);
 4  // 从键盘读取两个整数
 5  void swapValues(int variable1, int variable2);
 6  // 交换 variable1 和 variable2 的值
 7  void showResults(int output1, int output2);
 8  // 依次显示 variable1 和 variable2 的值
 9  int main()
10  {
     using namespace std;
11      int firstNum, secondNum;
12      getNumbers(firstNum, secondNum);
13      swapValues(firstNum, secondNum);
14      showResults(firstNum, secondNum);
15      return 0;
```

忘记在此处添加符号&

```
16 }
                                          忘记在此处添加符号&
17 void swapValues(int variable1, int variable2)
18 {
19     int temp;

20     temp = variable1;
21     variable1 = variable2;  <-------- 疏忽造成的局部变量
22     variable2 = temp;
23 }

24 <getNumbers 和 showResults 的定义和图 5.4 一样>
```

示范对话

```
Enter two integers: 5 10
In reverse order the numbers are: 5 10
```

自测题

7. 以下程序的输出是什么？

```
#include <iostream>
void figureMeOut(int& x, int y, int& z);
int main()
{
    using namespace std;
    int a, b, c;
    a = 10;
    b = 20;
    c = 30;
    figureMeOut(a, b, c);
    cout << a << " " << b << " " << c;
    return 0;
}

void figureMeOut(int& x, int y, int& z)
{
    using namespace std;
    cout << x << " " << y << " " << z << endl;
    x = 1;
    y = 2;
    z = 3;
    cout << x << " " << y << " " << z << endl;
}
```

8. 对于图 5.4 的程序，如果在 swapValues 的函数声明和函数头中，从第一个参数中删除符号&，该程序的输出是什么？注意，不要删除第二个参数的符号&。

9. 对于图 5.6 的程序，如果将 doStuff 函数的声明变成如下形式：

```
void doStuff(int par1Value, int par2Ref);
```

并相应地更改函数头，使形参 par2Ref 成为传值参数，那么程序的输出是什么？

10. 为名为 zeroBoth 的函数写一个 void 函数定义，该函数有两个 int 类型的传引用参数，函数作用是将两个变量的值设为 0。

11. 为名为 addTax 的函数写一个 void 函数定义。addTax 函数有参数：taxRate(用百分数表示的销售税率)和 cost(商品税前价格)。该函数要更改 cost 的值，在其中加上销售税。

12. 有返回值的函数能使用传引用参数吗？函数能否同时有传值和传引用参数？

5.3　使用过程抽象

我的记性太差了，
常常连自己的名字都忘记。

——米格尔·德·塞万提斯·萨维德拉，《唐吉诃德》

前面讲过，过程抽象的原则在于，函数应设计成像黑盒那样使用。程序员使用函数时，只需查看函数声明和说明函数用途的注释文字，不需要知道函数主体的任何细节。本节将围绕这个原则讨论多方面的主题。

在函数中调用其他函数

函数主体可包含对另一个函数的调用，这类似于在程序 main 部分(main 函数)中调用其他函数。唯一限制是函数使用前必须声明。如果像本书一直坚持的那样设置程序，将所有函数声明都放到 main 函数之前，将所有函数定义都放到 main 函数之后，就完全不必担心这个问题。虽然能在函数定义中调用函数，但不能在函数定义中嵌套另一个函数定义。

图 5.8 展示了图 5.4 的程序的增强版本。图 5.4 的程序总是交换 firstNum 和 secondNum 变量的值，而新程序在需要时才交换，它用 order 函数重新排列变量的值来满足以下条件：

```
firstNum <= secondNum
```

如果条件已经满足，firstNum 和 secondNum 变量不发生变化。如果 firstNum 大于 secondNum，就调用 swapValues 函数交换两个变量的值。顺序测试和交换变量值都在 order 函数主体内进行。所以，swapValues 函数在 order 函数主体内调用。这不会产生任何特别的问题。根据过程抽象原则，我们将 swapValues 函数视为执行一个行动(即交换两个变量的值)。这个行动在任何地方执行都是一样的。

图 5.8　在函数中调用其他函数

```
1  // 该程序演示一个函数调用另一个函数
2  #include <iostream>
3
4  void getInput(int& input1, int& input2);
5  // 从键盘读取两个整数
6
7  void swapValues(int& variable1, int& variable2);
8  // 交换 variable1 和 variable2 的值
9
10 void order(int& n1, int& n2);
11 // 对 n1 和 n2 变量中的值进行排序。函数调用后，n1 <= n2
12
13
14 void giveResults(int output1, int output2);
15 // 输出 output1 和 output2 中的值
16 // 假定 output1 <= output2
17
18 int main()
19 {
20     int firstNum, secondNum;
21
22     getInput(firstNum, secondNum);
```

```
23      order(firstNum, secondNum);
24      giveResults(firstNum, secondNum);
25      return 0;
26  }
27
28  // 使用 iostream:
29  void getInput(int& input1, int& input2)
30  {
31      using namespace std;
32      cout << "Enter two integers: ";
33      cin >> input1 >> input2;
34  }
35
36  void swapValues(int& variable1, int& variable2)
37  {
38      int temp;
39
40      temp = variable1;
41      variable1 = variable2;
42      variable2 = temp;
43  }
44
45  void order(int& n1, int& n2)        <-- 函数定义的顺序没有限制
46  {
47      if (n1 > n2)
48          swapValues(n1, n2);
49  }
50
51  // 使用 iostream:
52  void giveResults(int output1, int output2)
53  {
54      using namespace std;
55      cout << "In increasing order the numbers are: "
56           << output1 << " " << output2 << endl;
57  }
```

示范对话

```
Enter two integers: 10 5
In increasing order the numbers are: 5 10
```

前条件和后条件

为函数声明写注释时，一个好办法是将其分解为两种信息，即前条件和后条件。**前条件(precondition)**指出调用函数需满足什么条件。除非满足前条件，否则不要调用函数；强行调用不保证会正确执行。**后条件(postcondition)**描述函数调用结果。也就是说，在满足前条件的前提下，后条件指出函数执行后会达成什么条件。对于要返回值的函数，后条件应描述返回值。对于要对实参变量进行修改的函数，后条件应描述对实参进行的修改。

以图 5.8 的 swapValues 函数声明为例，它的注释可修改成以下形式：

```
void swapValues(int& variable1, int& variable2);
// 前条件：variable1 和 variable2 已被赋值
// 后条件：varible1 和 variable2 的值进行了交换
```

再以图 5.2 的 celsius 函数声明为例，它的注释可修改成以下形式：

```
double celsius(double fahrenheit);
// 前条件：fahrenheit 是用华氏度表示的温度值
// 后条件：返回等价摄氏度
```

假如唯一后条件就是对返回值的描述，程序员通常可以省略“后条件：”一词。上述函数声明注释可改成以下形式，完全能接受：

```
// 前条件：fahrenheit 是一个用华氏度表示的温度值
// 返回等价的摄氏度
```

以下函数声明给出了前条件和后条件的另一个例子：

```
void postInterest(double& balance, double rate);
// 前条件：balance 是非负的存储款账户余额，
// rate 是表示成百分数的利率，比如 5 表示 5%
// 后条件：使 balance 的值增加百分之 rate
```

不需要知道函数的定义就可以使用函数，所以这里只给出函数声明及其注释。

前条件和后条件不仅是总结函数行动的一种手段，还是设计和编写函数的第一步。先用注释描述程序要做什么，再在函数定义中指定具体如何做。如果发现不能以合理的方式实现设计规范，就回头重新思考函数应该做什么。无论如何，事先想清楚对函数的要求，既有助于减少设计错误，还能防止浪费时间写无用的代码。

有的程序员不喜欢在函数注释中使用“前条件”和“后条件”这两个词。但不管使用什么词，注释中始终应该包含前条件和后条件信息。

案例分析：超市定价系统

该案例分析解决一个很简单的编程任务。从表面看，对于如此简单的任务，包含的细节似乎多了一点。但以一个简单的任务为背景分析设计元素，能集中精力学习这些元素，不会被旁枝末节所困扰。一旦掌握了这个简单例子所展示的技术，就能举一反三，将同样的技术应用于更复杂的编程任务。

1. 问题定义

我们受 Quick-Shop 连锁超市委托写一个程序。它接收合理的输入，并确定一件商品的零售价格。定价策略是：对于预计能在一周内售出的任何商品，定价在批发价的基础上涨 5%；预计在货架上至少摆一周才能售出的任何商品，定价则以批发价为基础上涨 10%。必须注意：7 天内售出的商品使用一个较低的涨幅，即 5%；8 天后售出的商品则使用一个较高的涨幅，即 10%。程序必须在正确的时机从一种形式的计算转换为另一种形式。

和往常一样，必须清楚说明输入和输出。

- 输入：一件商品的批发价和预计多少天售出
- 输出：该商品的零售价

2. 问题分析

像对待许多简单的编程任务一样，我们把这个问题分解成三个主要的子任务。

(1) 输入数据。

(2) 计算商品零售价。

(3) 输出结果。

3 个子任务用 3 个函数实现，下面是这 3 个函数的函数声明及其注释。注意，只有需要由函数更改的参数才是传引用参数，其余参数是传值参数。

```
void getInput(double& cost, int& turnover);
// 前条件：用户准备好输入正确的值
// 后条件：cost 的值设为一件商品的批发价
// turnover 的值设为商品预计售出的天数

double price(double cost, int turnover);
// 前条件：cost 是一件商品的批发价
// turnover 是商品预计售出的天数
// 返回商品的零售价

void giveOutput(double cost, int turnover, double price);
// 前条件：cost 是一件商品的批发价
// turnover 是商品预计售出的天数，price 是商品的零售价
// 后条件：在屏幕上输出 cost，turnover 和 price 的值
```

有了函数声明之后，就可轻松写出程序 main 部分，如下所示：

```
int main()
{
    double wholesaleCost, retailPrice;
    int shelfTime;

    introduction();
    getInput(wholesaleCost, shelfTime);
    retailPrice = price(wholesaleCost, shelfTime);
    giveOutput(wholesaleCost, shelfTime, retailPrice);
    return 0;
}
```

虽然还没有写函数主体，而且对函数具体如何工作没有丝毫概念，但完全可以顺利地写出以上使用这几个函数的代码。这便是过程抽象原则的精髓，函数作为黑盒使用。

3. 算法设计

getInput 和 giveOutput 函数的实现非常简单，它们只由几个 cin 语句和 cout 语句构成。price 函数的算法由以下伪代码给出：

```
if turnover ≤ 7 days, then
    return (cost + cost 的 5%);
else
    return (cost + cost 的 10%);
```

4. 编码

程序要用到 3 个常量：一个低涨幅(5%)、一个高涨幅(10%)和低涨幅与高涨幅的分界线(7 天)。由于这些常量可能因公司定价策略而发生改变，所以在程序开始处将这 3 个常量定义成全局命名常量。使用了 const 修饰符的声明如下所示：

```
const double LOW_MARKUP = 0.05; // 低涨幅 5%
const double HIGH_MARKUP = 0.10; // 高涨幅 10%
const int THRESHOLD = 7; // 预计 7 天以内无法售出就使用 HIGH_MARKUP
```

price 函数主体的 C++代码可通过直接转换上述伪代码而得：

```
{
    if (turnover <= THRESHOLD)
        return ( cost + (LOW_MARKUP * cost) );
    else
```

```
        return ( cost + (HIGH_MARKUP * cost) );
    }
```

完整程序如图 5.9 所示。

图 5.9　超市定价系统

```
1  // 根据 Quick-Shop 连锁超市的定价策略来确定商品的零售价
2
3  #include <iostream>
4  const double LOW_MARKUP = 0.05; // 低涨幅 5%
5  const double HIGH_MARKUP = 0.10; // 高涨幅 10%
6  const int THRESHOLD = 7; // 如果预计 7 天内无法售出，则使用 HIGH_MARKUP
7
8  void introduction();
9  // 后条件：在屏幕上输出程序说明
10 void getInput(double& cost, int& turnover);
11 // 前条件：用户准备好输入正确的值
12 // 后条件：cost 的值设为一件商品的批发价
13 // turnover 的值设为商品预计售出的天数
14
15 double price(double cost, int turnover);
16 // 前条件：cost 是一件商品的批发价
17 // turnover 是商品预计售出的天数
18 // 返回商品的零售价
19 void giveOutput(double cost, int turnover, double price);
20 // 前条件：cost 是一件商品的批发价，turnover 是商品
21 // 预计售出的天数，price 是商品的零售价
22 // 后条件：在屏幕上输出 cost，turnover 和 price 的值
23
24 int main()
25 {
26     double wholesaleCost, retailPrice;
27     int shelfTime;
28     introduction();
29     getInput(wholesaleCost, shelfTime);
30     retailPrice = price(wholesaleCost, shelfTime);
31     giveOutput(wholesaleCost, shelfTime, retailPrice);
32     return 0;
33 }
34 // 使用 iostream:
35 void introduction()
36 {
37     using namespace std;
38     cout << "This program determines the retail price for\n"
39          << "an item at a Quick-Shop supermarket store.\n";
40 }
41 // 使用 iostream:
42 void getInput(double& cost, int& turnover)
43 {
44     using namespace std;
45     cout << "Enter the wholesale cost of item: $";
46     cin >> cost;
47     cout << "Enter the expected number of days until sold: ";
48     cin >> turnover;
49 }
50 // 使用 iostream:
51 void giveOutput(double cost, int turnover, double price)
52 {
53     using namespace std;
54     cout.setf(ios::fixed);
```

```
55     cout.setf(ios::showpoint);
56     cout.precision(2);
57     cout << "Wholesale cost = $" << cost << endl
58          << "Expected time until sold = "
59          << turnover << " days" << endl
60          << "Retail price = $" << price << endl;
61 }
62 // 使用已定义的常量 LOW_MARKUP，HIGH_MARKUP 和 THRESHOLD:
63 double price(double cost, int turnover)
64 {
65     if (turnover <= THRESHOLD)
66         return ( cost + (LOW_MARKUP * cost) );
67     else
68         return ( cost + (HIGH_MARKUP * cost) );
69
70 }
```

示范对话

```
This program determines the retail price for an item at a Quick-Shop
supermarket store. Enter the wholesale cost of item: $1.21
Enter the expected number of days until sold: 5
Wholesale cost = $1.21
Expected time until sold = 5 days
Retail price = $1.27
```

5．测试程序

测试程序时，一个重要策略就是测试所有种类的输入。虽然输入“种类”并没有精确定义，但在实际应用中，通常很容易判断一个程序能处理哪些种类的输入。目前这个超市定价程序主要有两种输入：使用 5%低涨幅的输入；使用 10%高涨幅的输入。因此，至少要测试两种情况：一是预计 7 天内售出商，二是预计 7 天以后才能售出商品。

另一种测试策略是测试边界值。遗憾的是，边界值也是一个笼统的概念。假如程序从一个输入(测试)值开始，行为就开始发生改变，就认为这个值是边界值。例如，在超市定价程序中，程序的行为在预计上架时间 7 天后发生改变，因此，7 就是边界值。对于 7 天或小于 7 天售出的商品和超过 7 天才能售出的商品，程序的行为是不一样的。所以，至少要测试刚好在 7 天时的程序的行为。通常还应测试刚好和边界值相差 1 时的行为，因为你在确定边界值时，很容易出现“相差 1”错误。所以，还应针对预计售出时间为 6 天和 8 天的情况测试这个程序。这是对前一段描述的测试输入的补充，即刚好比 7 天少一天和多一天。

√ 自测题

13. 一个函数定义能否在另一个函数定义的主体中出现？
14. 函数定义能否包含对另一个函数的调用？
15. 重写图 5.8 的 order 函数的函数声明注释，用前条件和后条件来描述。
16. 为预定义函数 sqrt 给出前条件和后条件，该函数返回其实参的平方根。

5.4　测试和调试函数

“我看见了那个怪物——我亲手造出来的怪物。”

——*玛丽·雪莱，《科学怪人》*

存根和驱动程序

视频讲解：*Stubs and Drivers*

每个函数都应独立于程序其他部分设计、编码和测试。这是自顶向下设计策略的精髓。将每个函数都视为独立单元，大型任务可转换为一组较小的、更易管理的子任务。但函数在目标程序外部如何测试？答案是写专门的测试程序。例如，图 5.10 展示了 getInput 函数的测试程序(函数在图 5.9 中使用)。像这样的程序称为**驱动程序**(driver)。驱动程序是临时工具，可以很简单。不需要包含精心设计的输入例程，也不需要执行最终程序所需的全部计算。唯一要做的就是以尽量简单的方式为函数获得参数值(通常从用户处获得)，然后执行函数并显示结果。如图 5.10 所示，一个循环允许连续使用不同的参数值测试函数，不必每次都重新运行程序。

单独测试每个函数，能发现程序中的大多数错误。另外，还便于判断具体哪个函数有错。相反，如果只是对整个程序进行测试，虽然也能发现错误，但或许不好判断错在哪里。更糟的是，或许以为自己知道错在哪里，但最后发现找错了地方。

测试好的函数可以在驱动程序中测试其他函数。每次测试一个函数，该函数必须是程序中唯一尚未测试的。但测试它时，完全可以使用其他已通过测试的函数。如出现 bug，必然是尚未测试的函数的 bug。例如，使用图 5.10 的驱动程序测试了 getInput 函数之后，就可以在驱动程序中将 getInput 用作输入例程来测试其他函数。

图 5.10　驱动程序

```
1  // getInput 函数的驱动程序
2  #include <iostream>
3
4  void getInput(double& cost, int& turnover);
5  // 前条件：用户准备好输入正确的值
6  // 后条件：cost 的值设为一件商品的批发价，
7  // turnover 的值设为商品预计售出的天数
8
9
10 int main()
11 {
12     using namespace std;
13     double wholesaleCost;
14     int shelfTime;
15     char ans;
16
17     cout.setf(ios::fixed);
18     cout.setf(ios::showpoint);
19     cout.precision(2);
20     do
21     {
22         getInput(wholesaleCost, shelfTime);
23
```

```
24         cout << "Wholesale cost is now $"
25              << wholesaleCost << endl;
26         cout << "Days until sold is now "
27              << shelfTime << endl;
28
29         cout << "Test again?"
30              << " (Type y for yes or n for no): ";
31         cin >> ans;
32         cout << endl;
33     } while (ans == 'y' || ans == 'Y');
34
35     return 0;
36 }
37 // 使用 iostream:
38 void getInput(double& cost, int& turnover)
39 {
40     using namespace std;
41     cout << "Enter the wholesale cost of item: $";
42     cin >> cost;
43     cout << "Enter the expected number of days until sold: ";
44     cin >> turnover;
45 }
```

示范对话

```
Enter the wholesale cost of item: $123.45
Enter the expected number of days until sold: 67
Wholesale cost is now $123.45
Days until sold is now 67
Test again? (Type y for yes or n for no): y
Enter the wholesale cost of item: $9.05
Enter the expected number of days until sold: 3
Wholesale cost is now $9.05
Days until sold is now 3
Test again? (Type y for yes or n for no): n
```

测试函数时，有时不得不用到其他尚未编写或测试的函数。这时可以使用其他函数的简化版本，称为**存根函数**(stub)。存根函数不必执行正确的计算，只需提供完成测试所需的值。而且由于非常简单，所以很容易编写。例如，图 5.11 的程序测试图 5.9 的 giveOutput 函数，并测试程序的基本布局。该程序要用到 getInput 函数，它已用图 5.10 的驱动程序进行了测试。还要用到 initializeScreen 函数，假定已用单独的驱动程序进行了测试(虽然没有列出)。最后，由于 price 函数尚未测试，所以要用存根函数取代。注意，即使 price 函数尚未写好，也能使用这个驱动程序。这样，在详细定义所有函数之前，就可完成对程序基本布局的测试。

图 5.11　含有存根函数的程序

```
1  // 根据 Quick-Shop 连锁超市的定价策略来确定商品的零售价
2
3  #include <iostream>
4  void introduction();
5  // 后条件：在屏幕上输出程序说明
6  void getInput(double& cost, int& turnover);
7  // 前条件：用户准备好输入正确的值
8  // 后条件：cost 的值设为一件商品的批发价
9  // turnover 的值设为商品预计售出的天数
10
11 double price(double cost, int turnover);
12 // 前条件：cost 是一件商品的批发价
13 // turnover 是商品预计售出的天数
14 // 返回商品的零售价
```

```
15  void giveOutput(double cost, int turnover, double price);
16  // 前条件：cost 是一件商品的批发价
17  // turnover 是商品预计售出的天数，price 是商品的零售价
18  // 后条件：在屏幕上输出 cost，turnover 和 price 的值
19
20  int main()
21  {
22      double wholesaleCost, retailPrice;
23      int shelfTime;

24      introduction();
25      getInput(wholesaleCost, shelfTime);
26      retailPrice = price(wholesaleCost, shelfTime);
27      giveOutput(wholesaleCost, shelfTime, retailPrice);
28      return 0;
29  }

30  // 使用 iostream:
31  void introduction() <------ 测试过的函数
32  {
33      using namespace std;
34      cout << "This program determines the retail price for\n"
35           << "an item at a Quick-Shop supermarket store.\n";
36  }
37  // 使用 iostream:
38  void getInput(double& cost, int& turnover) <------ 测试过的函数
39  {
40      using namespace std;
41      cout << "Enter the wholesale cost of item: $";
42      cin >> cost;
43      cout << "Enter the expected number of days until sold: ";
44      cin >> turnover;
45  }
                                              正在测试的函数
46  // 使用 iostream:
47  void giveOutput(double cost, int turnover, double price)
48  {
49      using namespace std;
50      cout.setf(ios::fixed);
51      cout.setf(ios::showpoint);
52      cout.precision(2);
53      cout << "Wholesale cost = $" << cost << endl
54           << "Expected time until sold = "
55           << turnover << " days" << endl
56           << "Retail price= $" << price << endl;
57  }

58  // 这只是一个存根函数：
59  double price(double cost, int turnover) <------ 存根函数
60  {
61      return 9.99; // 虽然不正确，但用于测试足够了
62  }
```

示范对话

```
This program determines the retail price for
an item at a Quick-Shop supermarket store.
Enter the wholesale cost of item: $1.21
Enter the expected number of days until sold: 5
Wholesale cost = $1.21
Expected time until sold = 5 days
Retail price = $9.99
```

使用由存根函数搭建的程序，可测试并逐渐充实基本程序框架，而不必在测试每个函数时都写全新的程序。因此，包含存根函数的程序通常是最有效的测试方法。一个常见的方案是用驱动程序测试某些基本函数，比如输入和输出函数，再用包含存根函数的程序测

试其他函数。每次都用一个完整函数替代存根函数。一旦测试好这个函数，就用另一个完整的函数替代另一个存根函数。依此类推，直至完成最终的程序。

> **函数测试的基本规则**
>
> 在程序中测试函数时，程序中的其他所有函数都应已通过了全面的测试和调试。

自测题

17. 函数测试的基本规则是什么？为什么说这是测试函数的理想方式？
18. 什么是驱动程序？
19. 为图 5.11 的 introduction 函数写驱动程序。
20. 为前面自测题 11 的 addTax 函数写驱动程序。
21. 什么是存根函数？
22. 为具有以下声明的函数写存根函数。不必写整个程序，只需写程序中使用的存根函数 (提示，这个存根函数应该非常短)：

```
double rainProb(double pressure, double humidity, double temp);
// 前条件：pressure 是以英寸汞柱为单位的气压
// humidity 是表示成一个百分数的相对湿度
// temp 是以华氏度为单位的温度
// 返回降雨概率，它是 0~1 之间的一个数字
// 0 表示绝对没机会降雨，1 表示百分之百会降雨
```

5.5　常规调试技术

视频讲解：*Debugging*

用存根函数和驱动程序精心测试，能检测出程序的许多 bug。但仅仅检查代码以及在各种测试情况下的输出，还不足以发现许多逻辑错误。针对这个问题，许多常规调试技术可供利用。

不抱成见

将系统作为整体来检查，不要固执地以为 bug 发生在特定位置。程序给出不正确的输出，应该检查源代码、不同测试情形中的输入和输出值以及算法背后的逻辑。以图 5.9 的超市定价系统为例，如显示错误价格，原因可能是因为输入值有别于你在测试情形中期待的值，造成程序明显出错。

一些新手程序员喜欢随意更改代码的某些部分，并指望这样就能修正错误。坚决杜绝这种做法！刚开始写简单程序时，这种技术或许有效。但对于较大的程序，它几乎必然会失败，而且经常造成新的错误。确认一处改动前，一定要确定自己理解了这一处改动在逻辑上造成的影响。

最后，如果老师允许，可将程序给别人看看。旁观者也许能很快发现当局者没有注意

到的错误。也可休息几小时，或第二天再检查程序，这样也许更容易找出错误。

检查常见错误

首先应检查常见错误，本书随处可见的“陷阱”和“编程提示”小节描述了许多这样的错误。常见错误的例子包括：①未初始化的变量；②“相差 1”错误；③超过数据边界；④自动类型转换；⑤该用==的地方用了=。

定位错误

确定 bug 的确切原因和位置，是修正错误最初几个步骤之一。检查不同测试情形的输入和输出行为，是定位错误的一种方式。与此相关的一个技术是在程序关键位置添加 cout 语句打印关键变量的值。cout 语句有时还可指出程序当前正在执行什么代码。第 3 章讲述循环时，曾用这种技术跟踪变量。但即使代码中没有循环，也可使用这种技术。

以图 5.12 的代码为例，它本来的目的是使用以下公式将华氏度换算为摄氏度：

$$C = \frac{5(F - 32)}{9}$$

图 5.12　有 bug 的温度换算程序

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      double fahrenheit;
7      double celsius;
8
9      cout << "Enter temperature in Fahrenheit." << endl;
10     cin >> fahrenheit;
11     celsius = (5 / 9) * (fahrenheit - 32);
12     cout << "Temperature in Celsius is " << celsius << endl;
13
14     retrun 0;
15 }
```

示范对话

```
Enter temperature in Fahrenheit.
100
Temperature in Celsius is 0
```

但在测试该程序时，输入 100 华氏度，输出“Temperature in Celsius is 0”。这显然不对，正确答案是 37.8 摄氏度。

为了跟踪错误，我们决定打印关键变量的值。本例似乎是换算公式出了问题，所以决定分两步调试。第一步计算(Fahrenheit - 32)；第二步计算(5 / 9)，然后输出这两个结果值。图 5.13 对此进行了演示。同时将原始行变成注释，办法是在行首添加//。这告诉编译器忽略原始行，但仍然把它留在程序中，以便稍后参考。还原代码删除//即可，不需要重新输入整行代码。

检查 cout 语句的结果，即可知道 bug 的准确位置。本例是换算系数的计算不正确。将换算系数设为 5/9，造成编译器执行整数除法运算，结果是零。简单修正办法是将其中一个操作数变成浮点型，从而执行浮点除法，而不是整数除法。例如：

```
double conversionFactor = 5.0 / 9;
```

确定 bug 后就可删除或注释掉调试代码。然后修改换算公式，获得程序的正确版本：

```
celsius = (5.0 / 9) * (fahrenheit - 32);
```

添加调试代码，用 cout 语句输出关键变量的值，这个简单的技术在几乎所有编程环境中都有用。但在程序中添加大量 cout 语句，有时可能非常繁琐。cout 语句的输出也许很长，难以解释。另外，添加大量调试代码可能造成新 bug。许多编译器和集成开发环境都提供了一个称为**调试器**(debugger)的独立程序，允许程序员在指定代码行——称为**断点**(breakpoint)——暂停执行，然后逐行执行后续代码。调试器逐行执行代码时，程序员可观察变量内容的变化，甚至能人工修改存储在变量中的值。所以，不需要 cout 语句即可查看关键变量的值。不同 C++开发环境中，调试器的界面、命令和功能也不同。所以，请查阅用户手册或咨询老师，了解具体如何使用这些功能。

图 5.13　用 cout 语句调试

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      double fahrenheit;
7      double celsius;
8
9      cout << "Enter temperature in Fahrenheit." << endl;
10     cin >> fahrenheit;
11
12     // 注释掉原始代码，但在
13     // 程序中保留以供参考
14     // celsius = (5 / 9) * (fahrenheit - 32);    ◄──── 注释掉的代码
15
16     // 添加 cout 语句验证(5 / 9)和 (fahrenheit - 32)
17     // 是否能得到正确的结果
18     double conversionFactor = 5 / 9;
19     double tempFahrenheit = (fahrenheit - 32);
20
21     cout << "fahrenheit - 32 = " << tempFahrenheit << endl;  ◄──── 用 cout 语句调试
22     cout << "conversionFactor = " << conversionFactor << endl;
23     celsius = conversionFactor * tempFahrenheit;
24     cout << "Temperature in Celsius is " << celsius << endl;
25
26     retrun 0;
27 }
```

示范对话

```
Enter temperature in Fahrenheit.
100
fahrenheit - 32 = 68
conversionFactor = 0
Temperature in Celsius is 0
```

assert 宏

5.3 节讨论了子程序的前条件和后条件。`assert`(断言)宏在 assert 语句的位置验证是否满足预期条件。条件不满足，程序显示错误消息并退出。为了使用 assert，首先要在程序中包含 assert 的定义：

```
#include <cassert>
```

为了使用 assert，请在需要强制断言的位置添加以下代码，提供你想求值为 true 的布尔表达式：

```
assert(boolean_expression);
```

assert 语句是一种**宏**，宏是与函数相似的结构。例如，以下函数用牛顿法计算数字 n 的平方根：

$$sqrt_{i+1} = \frac{1}{2}\left(sqrt_i + \frac{n}{sqrt_i}\right)$$

其中，$sqrt_0 = 1$，而且随着 i 趋于无穷大，$sqrt_i$ 就越接近 n 的平方根。用于实现该算法的函数要求 n 是正数，迭代次数(numIterations)也是正数。为了保证这个条件，可以在函数中添加 assert，如下所示：

```
// 使用牛顿法估算 n 的平方根
// 前条件：n 是正数，numIterations 是正数
// 后条件：返回 n 的平方根
double newtonSqroot(double n, int numIterations)
{
    double answer = 1;
    int i = 0;

    assert((n > 0) && (numIterations > 0)) ;
    while (i < numIterations)
    {
        answer = 0.5 * (answer + n / answer);
        i++;
    }
    return answer;
}
```

执行这个函数时，如果传递任何负数或零值作为参数，程序就会退出，并报告断言失败(assertion failed)。在需要断言的任何地方，都可采取类似方式使用 assert 语句，它是增强程序可靠性的一项出色技术。

发布应用程序时，可能不想在最终的可执行程序中包含 assert 语句，因为用户可能看到他们不明白的错误消息。如果在代码中添加了大量 assert 语句，删除起来可能有一定困难。幸好，只需在程序开头，在#include <cassert>语句之前添加以下语句，即可禁用所有 assert 宏：

```
#define NDEBUG
#include <cassert>
```

以后修改了程序并需要重新调试，删除#define NDEBUG 这一行，或者把它注释掉，即可重新启用全部 assert 语句。

√ 自测题

23. 对于语句：x = (x * y / z);，如何用 assert 宏防止除以 0 错误？
24. 什么是调试器？
25. 可以运用哪些常规技术来确定错误的根源？

小　结

- 程序所有子任务都可作为函数实现，要么实现成返回值的函数，要么实现成 void 函数。
- 形参是占位符，调用函数时由实参填充(替换)。有两种替换方式：传值调用和传引用调用。
- 传值调用替换形参的是实参的值；传引用调用实参应该是变量，替换形参的是变量本身而非变量的值。
- 要在函数定义中指定传引用参数，方法是在形参的类型名称后附加符号 &。
- 与传值参数对应的实参不会被函数调用更改，与传引用参数对应的实参可以被函数调用更改。函数要更改变量的值需使用传引用参数。
- 注释函数声明时，一个好办法是使用前条件和后条件。前条件指出函数调用时要满足什么条件。后条件描述函数调用结果。也就是说，在满足前条件的前提下，后条件指出函数执行后会达成什么条件。
- 测试某函数时，程序中的其他所有函数都应已通过了全面的测试和调试。
- 驱动程序是专门用来测试函数的程序。
- 存根函数是函数的简化版本。存根函数替代尚未测试(或尚未写好)的函数定义，先让程序的其余部分能进行测试。
- 利用调试器、cout 语句的巧妙放置以及 assert 宏对程序进行调试。

自测题答案

1.
```
Hello
Goodbye
One more time:
Hello
End of program.
```

2. 不需要。void 函数定义不要求包含 return 语句。void 函数定义可以但没必要包含 return 语句。

3. 图 5.2 的 initializeScreen 函数定义如省略 return 语句，绝对不会影响程序行为。程序仍可编译和运行，而且行为正常。类似地，省略 showResults 函数定义的 return 语句也不会影响程序行为。但省略 celsius 函数定义中的 return 语句就会产生严重错误，导致程序无法运行。区别在于，initializeScreen 和 showResults 是 void 函数，celsius 不是。

4.
```cpp
#include <iostream>

void productOut(int n1, int n2, int n3);
int main()
{
    using namespace std;
    int num1, num2, num3;
    cout << "Enter three integers; ";
    cin >> num1 >> num2 >> num3;
    productOut(num1, num2, num3);
    return 0;
}

void productOut(int n1, int n2, int n3)
{
    using namespace std;
    cout << "The product of the three numbers "
         << n1 << ", " << n2 << ", and "
         << n3 << " is " << (n1*n2*n3) << endl;
}
```

5. 具体答案和你使用的系统有关。

6. 后跟分号的 void 函数调用是语句。相反，若函数有返回值(非 void 函数)，则函数调用是表达式。

7.
```
10 20 30
1 2 3
1 20 3
```

8.
```
Enter two integers: 5 10
In reverse order the numbers are: 5 5
```
← 此处有区别

9.
```
par1Value in function call = 111
par2Ref in function call = 222
n1 after function call = 1
n2 after function call = 2
```
← 此处有区别

10.
```cpp
void zeroBoth(int& n1, int& n2)
    {
        n1 = 0;
        n2 = 0;
    }
```

11.
```cpp
void addTax(double taxRate, double& cost)
{
    cost = cost + ( taxRate / 100.0 ) * cost;
}
```

注意： 本题要求将 taxRate 表示成百分数(而不是小数)。所以，除以 100 就可将百分数转换成小数。

例如 10%相当于 10/100.0，或者说一成销售税。

12. 可以。要返回值的函数可以使用传引用参数。另外，函数可混合使用传值和传引用参数。
13. 不能，函数定义不能放到另一个函数定义内部。
14. 是的，一个函数定义可包含对另一个函数的调用。
15.
```
void order(int& n1, int& n2);
// 前条件：变量 n1 和 n2 已被赋值
// 后条件：n1 和 n2 的值重新排序，使 n1 <= n2
```
16.
```
double sqrt(double n);
// 前条件：n >= 0
// 返回 n 的平方根
```

如果愿意，可以像下面这样重写第二个注释行，添加“后条件：”。但是，如果函数只是返回一个值，上述形式才是最常用的。

```
// 后条件：返回 n 的平方根
```

17. 函数测试的基本规则是，在程序中测试函数时，该函数必须是该程序中唯一尚未测试的函数，其他所有函数都必须进行全面测试和调试。之所以说它是测试函数的理想方式，是因为假如遵循这个规则，那么一旦发现 bug，就知道它肯定是当前测试的那个函数的 bug。
18. 驱动程序是专为测试一个函数而设计的程序。
19.
```
#include <iostream>

void introduction();
// 后条件：在屏幕上输出程序说明
int main()
{
    using namespace std;
    introduction();
    cout << "End of test.\n";
    return 0;
}
// 使用 iostream:
void introduction()
{
    using namespace std;
    cout << "This program determines the retail price for\n"
         << "an item at a Quick-Shop supermarket store.\n";
}
```
20.
```
// addTax 函数的驱动程序
#include <iostream>

void addTax(double taxRate, double& cost);
// 前条件：taxRate 是销售税的百分数，cost 是一件商品的税前价格
// 后条件：修改 cost 的值，在其中加上销售税

int main()
{
    using namespace std;
    double cost, taxRate;
    char ans;

    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    do
    {
        cout << "Enter cost and tax rate:\n";
        cin >> cost >> taxRate;
        addTax(taxRate, cost);
```

```
        cout << "After call to addTax\n"
             << "taxRate is " << taxRate << endl
             << "cost is " << cost << endl;

        cout << "Test again?"
             << " (Type y for yes or n for no): ";
        cin >> ans;
        cout << endl;
    } while (ans == 'y' || ans == 'Y');

    return 0;
}

void addTax(double taxRate, double& cost)
{
    cost = cost + ( taxRate/100.0 ) * cost;
}
```

21. 存根函数是函数的简化版本，用于临时替代该函数，确保其他函数能顺利开始测试。

22.
```
// 这只是存根函数
double rainProb(double pressure, double humidity, double temp);
{
    retrun 0.25; // 虽然不正确，但用来测试足够了
}
```

23. `assert (z != 0);`

24. 调试器是一种工具，允许程序员设置断点，逐行执行代码，以及查看或修改变量值。

25. 抛弃成见，添加 cout 语句来排除错误，使用调试器，查找常见错误，以及设计多样化的测试，所有这些都可有助于调试程序。

编 程 练 习

编程练习一般只需写很小的程序，运用本章提到的编程概念。

1. 写函数计算 4 个测验分数的平均数和标准差。标准差等于以下 4 个值的平均值的平方根：$(s_i\text{-}a)^2$。其中，a 是 4 个测验分数(s_1 ，s_2 ，s_3 和 s_4)的平均数。例如，假定 4 个分数是 5，6，8 和 9，则它们的平均数是 7，而标准差等于$((5-7)^2+(6-7)^2+(8-7)^2+(9-7)^2)/4$ 的平方根，即 1.5811。函数获取 6 个参数，还会调用另外两个函数。将该函数嵌入一个驱动程序，以便重复测试该函数，直到要求程序终止。

2. 写程序读入用英尺和英寸表示的长度，输出米和厘米长度。至少使用 3 个函数：一个用于输入，一个(或多个)用于计算，另一个用于输出。在程序里包含一个循环，允许用户输入新值来重复计算，直到表明要终止程序。1 英尺等于 0.3048 米，1 米等于 100 厘米，1 英尺等于 12 英寸。

3. 写一个程序，和上个练习相似，将米和厘米换算为英尺和英寸。用函数执行子任务。

4. (先完成前两个练习)写程序合并前两个编程练习中的函数。先询问是想把英尺和英寸换算为米和厘米，还是相反。然后执行换算。要求用户输入整数 1 指定一种换算，输入整数 2 指定另一种。程序读取用户输入，然后执行一个 `if-else` 语句。`if-else` 语句的每个分支都是一个函数调用。对于 `if-else` 语句调用的这两个函数，它们的函数定义和前两个练习很相似。所以，考虑在函数主体中调用其他函数。用循环允许用户输入新值来重复计算，直到要求终止程序。

5. 视频讲解：*Solution to Practice Program 5.5*

   写程序读入磅和盎司重量，输出克和克重量。至少使用 3 个函数：一个用于输入；一个(或多个)用于计算；另一个用于输出。用循环允许用户根据输入新值来重复计算，直到要求终止程序。1 千克等于 2.2046 磅，1 千克等于 1000 克，1 磅等于 16 盎司。

6. 写和上个练习相似的程序，将千克和克重量换算为磅和盎司重量。用函数执行子任务。

7. (先完成前两个练习)写程序合并前两个编程练习中的函数。先询问是想把磅和盎司换算为千克和克，还是相反。然后执行换算。要求用户输入整数 1 指定一种换算，输入整数 2 指定另一种。程序读取用户的输入，然后执行一个 if-else 语句。if-else 语句的每个分支都是一个函数调用。对于 if-else 语句调用的这两个函数，它们的函数定义和前两个练习很相似。所以，考虑在函数主体中调用其他函数。用循环允许用户输入新值来重复计算，直到要求终止程序

8. (先完成练习 4 和 7)写程序合并编程练习 4 和 7 的函数。先询问是想进行长度换算还重量换算。如果选择长度，询问是想把英尺和英寸换算为米和厘米，还是相反。如果选择重量，则询问是想把磅和盎司换算为千克和克，还是相反。然后执行换算。要求用户输入整数 1 指定一种换算，输入整数 2 指定另一种。程序读取输入，然后执行一个 if-else 语句。if-else 语句的每个分支都是一个函数调用。对于 if-else 语句调用的这两个函数，它们的函数定义和练习 4 和 7 很相似，所以可以在函数主体中调用其他函数。但可以很容易地修改练习 4 和 7 的程序来获得新的函数定义。

   注意，新程序会在 if-else 语句中嵌套 if-else 语句，但这种嵌套是间接的。外层 if-else 语句包含两个函数调用作为分支，这两个函数调用各自包含一个 if-else 语句。但其实不必关心嵌套问题。它们只是简单的函数调用，具体细节都隐藏在黑盒中(黑盒是在你定义这些函数时创建的)。如果你的思路是创建一个 4 路分支，就可能错了。只需考虑两路分支(即使整个程序最终会分支为 4 路)。用循环允许用户输入新值来重复计算，直到要求终止程序

9. 任意三角形的面积计算公式：

$$\text{面积} = \sqrt{s(s-a)(s-b)(s-c)}$$

   其中，a，b 和 c 是三角形的 3 个边长，s 是半周长：

$$s = (a+b+c)/2$$

   写 void 函数根据 3 个边长计算三角形面积和周长(而不是半周长)。获取 5 个参数：其中 3 个传值，指定 3 个边长；另外 2 个传引用，存储计算好的面积和周长。将函数变得更健壮。注意，并非 a，b 和 c 的任意组合都能产生三角形。函数应该为有效数据生成正确结果，为无效组合显示合理提示。

编 程 项 目

编程项目要求综合运用多方面的知识来解决问题，程序一般比编程练习大，解题方式多样化。

1. 写程序将 24 小时制转换为 12 小时制，例如 14:25 转换为 2:25 PM。输入两个整数。程序至少要有 3 个函数：一个用于输入，一个执行转换，另一个用于输出。将 AM/PM 信息作为 char 值记录，用'A'表示 AM，用'P'表示 PM。所以，用于执行转换的函数有一个 char 类型的传引用参数，用于记录是 AM 还是 PM(该函数应该还有其他参数)。使用循环允许用户输入新值来重复计算，直到要求终止。

2. 写程序要求用户输入当前时间和等待时间。时间用两个整数输入，一个代表小时数，一个代表分钟数。程序输出过了等待时间后的时间。时间采用 24 小时制。使用循环允许重复计算，直到要求终止。

3. 修改编程项目 2 的程序来使用 12 小时制，比如 3:45 PM。

4. 写程序针对任意金额的零钱(从 1 美分到 99 美分)，算出应该给出多少个面值分别为 25 美分(1 quarter)、10 美分(1 dime)和 1 美分(1 penny)的硬币。假设输入金额为 86 美分，则输出应该是：

```
86 cents can be given as
3 quarter(s) 1 dime(s) and 1 penny(pennies)
```

   注意：本程序只使用面值为 25 美分、10 美分和 1 美分的硬币，不使用面值为 5 美分(1 nickel)和 50 美分(half-dollar)的硬币。程序至少要使用下面这个函数(还有其他函数)：

```
void computeCoins(int coinValue, int& num, int& amountLeft);
// 前条件：coinValue 是一种硬币的面值，0 < coinValue < 100;
// amountLeft 是剩余的金额(美分数)，0 <= amountLeft < 100
// 后条件：在目前剩余 amountLeft 美分的前提下，num 被设为
// 最多能凑出多少个面值为 coinValue 美分的硬币
// 然后，amountLeft 递减这些硬币的总金额，也就是递减 num * coinValue
```

例如，假设变量 amountLeft 的值为 86，在执行以下调用之后：

```
computeCoins(25, number, amountLeft);
```

number 的值变成 3，而 amountLeft 的值变成 11，因为如果从 86 美分里扣除 3 个 quarter(每个 quarter 面值是 25 美分)，则剩余 11 美分。使用循环允许用户输入新值来重复计算，直到要求终止。(**提示**：使用整数除法和%操作符实现该函数。)

5. 天气较冷时，气象学家会报告一种指数，称为风冷力指数[①]。风冷力指数(W)根据以下公式估算：

$$W = 13.12 + 0.6215 * t - 11.37 * v^{0.16} + 0.3965 * t * v^{0.016}$$

其中：

v——风速，单位是 m/s(米 / 秒)

t——气温，单位是℃(摄氏度)，$t <= 10$。

W——风冷力指数，单位是℃(摄氏度)。

写函数返回风冷力指数。代码应确保不违反气温限制。到学校图书馆查过期报纸的天气预报，把你算出的风冷力指数与报纸上报告的进行比较。

6. 视频讲解：*Solution to Programming Project 5.6*

在 Puzzlevania 大陆上，Aaron，Bob 和 Charlie 为他们谁才是最伟大的 puzzler 争论不休。为了一劳永逸地结束这场争吵，他们决定来一场至死方休的决斗。Aaron 枪法最差，只有 1/3 的概率射中目标。Bob 稍好一点，射中目标的概率是 1/2。Charlie 则是神枪手，百发百中。射中意味着死亡，被射中的人退出决斗。

为了弥补由于枪法差异而造成的不公平，他们决定按顺序开枪，首先开枪的是 Aaron，其次是 Bob，最后是 Charlie。然后一直这样重复，直到最后只有一个人站着，这个人将成为有史以来最伟大的 puzzler。

a. 写函数模拟一次射击的情况。它应当使用以下声明：

```
void shoot(bool& targetAlive, double accuracy);
```

它模拟某人在给定命中率(accuracy)的情况下朝 targetAlive 射击的情况。具体做法是生成一个 0～1 的随机数。如随机数小于 accuracy，则目标被射中，targetAlive 应被设为 false。本书第 4 章描述了如何生成随机数。

例如，假定 Bob 向 Charlie 开枪，那么具体函数调用如下：

```
shoot(charlieAlive, 0.5);
```

在这个例子中，charlieAlive 是布尔变量，指出 Charlie 是否存活。继续步骤 b 之前，用一个驱动程序测试好你的函数。

b. 一个显而易见的战术是，每个人都朝当前仍然存活的枪法最好的射手开枪，该射手最致命，要尽快“干掉”。写第二个函数 startDuel，基于该战术用 shoot 函数模拟一次完整的决斗。应该一直循环，直到只剩下一个参加者。调用 shoot 函数时要指定正确的目标，并依据当前是谁正在开枪，指定正确的命中率。函数应返回一个变量，指出谁是最后的胜利者。

c. 在 main 函数中调用 1000 次 startDuel 函数，跟踪记录每个参加者获得胜利的次数。输出每个参加者最终获得胜利的概率(前提是每个人都使用步骤 b 的战术，朝当前枪法最好的人开枪)。

d. 一个违反直觉的战术是 Aaron 在他第一次开枪时故意失手。之后，每个人仍然使用步骤 b 的战术，朝当前枪法最好的人开枪。采用这个战术，Aaron 保证能在第一轮射击中存活，因为 Bob 和 Charlie 会相互射击。修改程序以适应新战术，输出每个参加者的获胜概率。

7. 写程序输入一个日期(例如 July 4，2008)，输出那个日期是星期几。以下算法基于 *http://en.wikipedia.org/wiki/Calculating_the_day_of_the_week*。你的实现需用到几个函数：

---

① 也称为寒冷指数，用于表征人体皮肤温度为 33℃时人体表面的散热量，反映了在室外寒冷环境下，风速与气温对裸露皮肤的影响程度。——译注

```
bool isLeapYear(int year);
```

leap year 是闰年的意思。如 year 是闰年，函数返回 true，否则返回 false。以下是判断是否为闰年的伪代码：

```
leapYear = ((能被 400 整除的年份) or (能被 4 整除，但不能被 100 整除的年份))
```

```
int getCenturyValue(int year);
```

这个函数获取年份的头 2 位(也就是世纪值)，除以 4，再取余数。用 3 来减这个余数，结果乘以 2 并返回。例如，假定年份是 2008，那么：20/4 得 5 余 0。3–0=3。最后返回 3*2=6。

```
int getYearValue(int year);
```

这个函数判断一个年份相对于本世纪元年过了多少年，并相应地计算出一个值。首先，提取年份的最后两位。以 2008 为例，首先提取出 08。接着，考虑闰年的因素。具体的办法是用上一步的值除以 4，并丢弃余数。最后，两个结果加到一起并返回。例如，从 2008 中我们提取出 08。然后，8/4 得 2 余 0。最后返回 8 + 2 = 10。

```
int getMonthValue(int month, int year);
```

这个函数参照下表返回一个值，注意需要在函数中调用 isLeapYear 函数。

月份	返回值
January	0(闰年返回 6)
February	3(闰年返回 2)
March	3
April	6
May	1
June	4
July	6
August	2
September	5
October	0
November	3
December	5

最后，为了计算是星期几，需要计算以下几个值之和：给定日期中的“日”(例如，7 月 4 日就是 4)以及 getMonthValue、getYearValue 和 getCenturyValue 这三个函数的返回值。结果除以 7 并计算余数。余数为 0 对应 Sunday，1 对应 Monday……6 对应于 Saturday。例如，July 4，2008 这个日期应该计算成：(4)+(getMonthValue) + (getYearValue) + (getCenturyValue) = 4 + 6 + 10 + 6 = 26。26/7 得 3 余 5，所以答案是 Friday(周五)。

你的程序应该允许用户输入一个日期，并输出这个日期是星期几，一切都用英语表示。

程序应包含一个名为 getInput 的 void 函数，它提示用户输入日期，并以传引用参数的形式返回月、日和年。可选择让用户决定是以数字(1~12)还是月份名称来输入月份。

8. 完成上个编程项目，创建顶级函数 dayOfWeek，它的声明如下所示：

```
int dayOfWeek(int month, int day, int year);
```

函数要封装必要的逻辑，以 int 的形式返回指定日期是星期几(Sunday = 0，Monday = 1，依此类推)。要添加校验代码检测无效的输入。对于无效的输入，函数应返回-1。在 main 函数中写一个测试驱动程序检查 dayOfWeek 是否返回正确的值。测试时，至少要有两个测试情形提供无效的输入。

9. 写程序玩简化版的黑杰克(21 点)游戏。不是从一墩牌中选，而是掷一个 14 面的骰子来决定一张牌。规则如下。

(1) 抽一次牌随机选择 2~10，J，Q，K 或 A。A 记为 11 点。J，Q 和 K 均记为 10 点。换言之，一次

抽牌从 2，3，4，5，6，7，8，9，10，10，10，10，11 这些值中随机选一个。

(2) 玩家输入赌注。赌注必须是正数，而且不得超过玩家当前持有的赌金。

(3) 玩家抽两张牌。由计算机充当的庄家抽一张。庄家明牌。

(4) 玩家点数为 22 点或更大，则玩家“爆牌”，失去赌注，本轮结束。

(5) 玩家点数为 21 点或更小，可选择“要牌”或“停牌”。如玩家选择“要牌”，则玩家再抽一张牌，返回步骤 4。如玩家选择“停牌”，则继续步骤 6。

(6) 现在轮到庄家。庄家再抽一张牌。庄家点数小于 17 则必须“要牌”。庄家点数为 22 或更大，则庄家“爆牌”，玩家赢得赌注，本轮结束。庄家必须一直要牌，直到“爆牌”或取得 17 到 21 之间的点数。

(7) 如庄家未“爆牌”，就和玩家比较点数。点数一样是“平局”，玩家不输不赢。庄家点数比玩家点数大，则玩家失去赌注。玩家点数比庄家点数大，则玩家赢得赌注。

程序允许玩家重复玩，直到要求退出或用光赌金。假定玩家初始赌金是 100 元。

该程序用“自下而上”方法开发。先写驱动程序测试函数，再写更高级代码实现 21 点逻辑。最起码要从以下函数开始：

`int rollDice();`	掷点 2~10，J，Q，K 或 A 并返回值
`int getWager(int money)`	money 参数是玩家当前持有赌金金额。函数要求玩家从键盘输入此次下注金额并返回该金额。函数应确保输入金额有效(正数且<= money)。如输入无效金额，提示玩家重新输入

用驱动程序测试好这些函数后，在抽象链中上移并实现以下函数。

`void playerTurn(int money,`
`int &wager,`
`bool &bust,`
`int &total,`
`int &houseRoll)`
	实现轮到玩家时的 21 点逻辑(步骤 2~5)。money 参数是玩家当前持有赌金金额。wager 返回玩家下注金额。bust 在玩家爆牌时返回 true，否则返回 false。total 返回玩家总点数。houseRoll 返回庄家所抽第一张牌的点数(步骤 3)。在调用 playerTurn 的代码中(而不是在 playerTurn 内部)测试本轮是否结束，并更新玩家当前持有赌金金额

`void houseTurn(int houseRoll,`
`bool &bust,`
`int &total`
	实现轮到计算机庄家时的逻辑(步骤 6)。houseRoll 是庄家明牌的那张牌的点数。bust 在庄家爆牌时返回 true，否则返回 false。total 返回庄家总点数。在调用 houseTurn 代码中(而不是在 houseTurn 内部)测试本轮赢家(步骤 7)

测试这些函数确保正确工作。最后写整个 21 点的逻辑。如果愿意，可以再写额外的函数。

10. 重做编程项目 9，改为用“自顶向下”方法开发。也就是说，先写 main 函数，留好 playerTurn 和 houseTurn 的存根。为了简单地实现存根，同时保留测试时的灵活性，一个方法是引用变量的值直接从键盘输入。例如，为模拟玩家下注 50 元，庄家抽了一张 6，玩家当前总点数 15，函数向 wager 输入 50，将 bust 设为 false，向 total 输入 15，并向 houseRoll 输入 6。main 函数的逻辑搞定后，再用 rollDice 和 getWager 的存根实现 playerTurn 和 houseTurn 中的逻辑。同样地，从键盘输入值最简单。搞定 playerTurn 和 houseTurn 后，再实现最底层的 rollDice 和 getWager 逻辑来完成程序。

第 6 章　I/O 流——对象和类入门

鱼言坐拥溪水与池塘，然除此存否胜处别疆？

——鲁珀特·布鲁克，《天堂》(1913)

溪水中的树叶，不知归于湖泊还是大海；流中的输出，不知归于屏幕还是文件。[①]

——某校计算机系厕所墙壁(1995)

概述

I/O 指程序输入(Input)和输出(Output)。输入可来自键盘或文件。类似地，输出可发送给屏幕或文件。本章解释如何写程序从文件获取输入，再将输出发送给另一个文件。

输入通过一个名为“流”的 C++构造传给程序，程序输出也通过“流”传给某个输出设备。“流”是我们的第一个“对象”例子。对象乃是一种特殊变量，拥有自己的专用函数。某种意义上，这些函数是“附加”到变量上的。处理对象是 C++语言的重要功能。正是因为这个功能，才使其和早期编程语言有了根本性区别。本章介绍什么是流，并解释如何将其用于程序 I/O。在解释流的过程中，要指引你熟悉对象的一些基本思路，并解释怎样在程序中使用对象。

预备知识

本章基于第 2 章～第 5 章的知识。

6.1　流和基本文件 I/O

我的老天啊！我说了四十多年的散文，竟然一点儿也不知情……

——莫里哀，《贵人迷》

前面一直在用文件保存程序。此外，还可用文件保存程序输入或接收程序输出。用于程序 I/O 的“文件”和用于保存程序的“文件”没有区别。利用即将讨论的“流”，程序可以采用统一的方式处理文件输入与键盘输入，并可采用统一的方式处理文件输出与屏幕输出。

流(stream)是由字符(或其他类型的数据)构成的“流”(flow)。流向程序，称为**输入流**(input stream)。流出程序，则称为**输出流**(output stream)。如输入流来自键盘，表明程序要从键盘获取输入。如输入流来自文件，表明程序要从文件获取输入。类似地，输出流可发送给屏幕或文件。

虽然你可能还没有意识到，但以前的程序事实上已经使用了流。以前用过的 cin 是连接到键盘的输入流，而 cout 是连接到屏幕的输出流。这两个流由系统自动提供给你的程序，唯一要求就是在程序中使用 include 指令包含头文件 iostream。除此之外，还可定义来自文件的流，或者发送给文件的流。定义好之后，就可采用与使用 cin 和 cout 相同的方式使用它们。

① 溪水和流在英语中都是 stream。——译注

例如，假定已定义好名为 inStream 的输入流，这个流来自某文件(马上就要解释具体如何定义)。执行以下语句，就可使用来自该文件的一个数填充 int 变量 theNumber：

```
int theNumber;
inStream >> theNumber;
```

类似地，假定已定义好名为 outStream 的输出流，这个流发送给另一个文件，就可将上述变量的值发送给那个文件。以下语句输出字符串"theNumber is "，后跟变量 theNumber 的值，所有输出都发送给与 outStream 流连接的文件：

```
outStream << "theNumber is " << theNumber << endl;
```

一旦流连接到希望的文件，程序就可以像执行键盘和屏幕 I/O 那样执行文件 I/O。

为什么要用文件来 I/O

前面所用的键盘输入和屏幕输出都只是和临时数据打交道。程序终止，用户从键盘输入的数据和屏幕上显示的数据丢失。文件则提供了一种永久保存数据的方式。文件中的内容会保留下来，直到有人或程序改变了文件。如程序将输出发送给文件，那么在程序结束运行之后，输出文件仍会保留。输入文件则可供多个程序反复使用，不需要单独为每个程序输入数据。

程序所用的输入和输出文件是同一种类型的文件，这种文件可用一个编辑器(比如用来写代码的编辑器)读写。所以，可在方便时用编辑器为自己的程序创建输入文件，或打开程序生成的输出文件，而非只能在程序运行时读写。

文件还提供了处理大量数据的方便途径。通过从较大的输入文件获取输入，程序能快速获取大量数据，不需要用户进行大量的键盘输入。

文件 I/O

程序从文件获取输入，称为**读取**文件；将输出发送给文件，称为**写入**文件。虽然还有其他方式可从文件中读取输入，但我们采用的是从头到尾读取(或者说一直读到文件尾)。采用这种方式，程序就不能倒退回去读取文件中已经读取过的任何内容。这类似于程序从键盘输入，所以你不应感到陌生，也不应感到怪异(如后文所述，程序可从文件头开始重新读取文件，但这是“从头读取”，而非“倒退读取”)。类似地，采用这种方式，程序要从文件头开始写入，并连续地写入。程序不能倒退回去，更改以前写入文件的任何输出。这类似于将输出发送到屏幕。可以向屏幕发送不断增多的输出，但不能退回去更改以前的输出。程序为了从文件获取输入，或者将输出发送给文件，必须借助流来连接文件。

在 C++中，流是一种称为“对象”的特殊变量。下一节将讨论对象，本节首先要描述程序如何用流对象来进行简单文件 I/O。要用流从文件获取输入(或者将输出发送给文件)，首先必须声明流，而且必须将流连接到文件。

可将流连接到的文件视为这个流的值。可以将流与文件断开，再连接到另一个文件，换言之，可以更改流变量的值。然而，必须使用专门处理流的函数才能完成这些更改。不能像使用 int 或 char 类型的变量那样在赋值语句中使用流变量。虽然流是变量，但它们是特殊的变量。

cin 和 cout 流是系统已经为你声明好的。要将流连接到文件，必须像声明其他任何变量那样声明流。“输入文件流”(input-file stream)变量的类型名称是 ifstream；“输出文件流”(output-file stream)变量的类型名称是 ofstream。所以，要将 inStream 声明为用于文件的输入流，将 outStream 声明为用于另一个文件的输出流，需要使用如下所示的语句：

```
ifstream inStream;
ofstream outStream;
```

ifstream 和 ofstream 类型在头文件为 fstream 的库中定义。所以，任何程序要想以这种方式声明流，都必须包含以下预编译指令(通常放在接近文件开头的位置)：

```
#include <fstream>
```

使用 ifstream 和 ofstream 类型时，程序还必须包含以下语句：

```
using namespace std;
```

该语句一般放在接近文件开头的位置，或者放在使用了 ifstream 或 ofstream 类型的函数体开头。

流变量(例如前面声明的 inStream 和 outStream)必须**连接到**一个文件。这称为**打开文件**，用 open 函数完成该操作。例如，假定希望输入流 inStream 连接到 infile.dat 文件，程序首先执行以下语句，然后才能从该文件读取输入：

```
inStream.open("infile.dat");
```

对于函数调用，这似乎是一种怪异的语法。下一节将更全面地解释这种特殊语法。就目前来说，对于 open 函数调用的写法，你只需注意两个细节。首先，流变量名称和一个圆点(也就是句点符号)要放在函数名称 open 之前，文件名则被指定为 open 函数的参数。其次，文件名要使用一对双引号。如输入文件和你的程序在同一个目录，像这样写或许就足够了。但某些情况下，可能还要指定文件所在的目录。具体如何指定目录因系统而异。如需指定目录，请询问你的老师或其他懂行的人，请他们解释一下细节。

一旦声明好输入流变量，并用 open 函数将其连接到文件，程序就可使用提取操作符>>从文件获取输入。例如，以下语句从连接到 inStream 的文件读取两个数，将它们分别放入变量 oneNumber 和 anotherNumber 中：

```
int oneNumber, anotherNumber;
inStream >> oneNumber >> anotherNumber;
```

输出流采用和输入流相同的方式来打开(也就是连接到)文件。例如，以下语句声明输出流 outStream 并将其连接到 outfile.dat 文件：

```
ofstream outStream;
outStream.open("outfile.dat");
```

针对 ofstream 类型的一个流，成员函数 open 会新建一个尚不存在的输出文件。如输出文件存在，成员函数 open 会丢弃文件内容。所以在调用 open 函数之后，输出文件为空。

一旦调用 open 将文件连接到 outStream 流，程序就可使用插入操作符<<将输出发送到那个文件。例如，以下语句将两个字符串和两个变量 oneNumber 与 anotherNumber 的内容写入和 outStream 流连接的文件(本例是 outfile.dat 文件)：

```
outStream << "oneNumber = " << oneNumber
```

```
                    << " anotherNumber = " << anotherNumber;
```

注意程序在处理文件时，文件好像有两个名称。一个是操作系统使用的标准文件名，即**外部文件名**。示范代码的外部文件名是 infile.dat 和 outfile.dat。某种意义上，外部文件名才是文件的“真实名称”，是操作系统使用的名称。这些外部文件名的拼写规范因系统而异；你可从老师或其他专家那里了解这些规范。在你的系统上，本例所用的 infile.dat 和 outfile.dat 可能像是合法文件名，也可能不像是。无论如何，应遵循自己的系统的规范来命名文件。虽然外部文件名是文件的真实名称，但它通常只在程序中使用一次。外部文件名作为 open 函数的参数使用。一旦打开了文件，为了继续引用该文件，就必须使用与文件连接的那个流的名称。所以，在程序中，流对象的名称相当于文件的第二个名称。

图 6.1 的程序从文件读取 3 个数，将它们的总和与其他文本写入另一个文件。

> **一个文件两个名称**
>
> 程序使用的每个输入和输出文件都有两个名称。**外部文件名**是文件真实名称，但只在 open 函数调用中使用一次，该函数将文件连接到一个流。一旦调用了 open，就必须将流名称作为文件名使用。

图 6.1　简单文件输入 / 输出

```
1  // 从文件 infile.dat 读取 3 个数，求和，将结果写入文件 outfile.dat
2  // (该程序的一个更好的版本将在图 6.2 给出)
3
4  #include <fstream>
5  int main()
6  {
7      using namespace std;
8      ifstream inStream;
9      ofstream outStream;
10
11     inStream.open("infile.dat");
12     outStream.open("outfile.dat");
13     int first, second, third;
14     inStream >> first >> second >> third;
15     outStream << "The sum of the first 3\n"
16               << "numbers in infile.dat\n"
17               << "is " << (first + second + third)
18               << endl;
19     inStream.close();
20     outStream.close();
21     return 0;
22 }
```

infile.dat (不被程序改变)	outfile.dat (程序运行后)
1 2 3 4	The sum of the first 3 numbers in infile.dat is 6

没有屏幕输出，也没有键盘输入

程序完成从文件输入或者向文件输出之后，应该将文件关闭。关闭文件导致流与文件断开。调用 close 函数关闭文件。图 6.1 的以下代码展示了如何使用 close 函数：

```
inStream.close();
outStream.close();
```

注意 close 函数不获取任何参数。如程序正常终止，但没有关闭文件，系统会自动为你关闭。但最好养成主动关闭文件的好习惯，原因有二。其一，只有在程序正常终止的前提下，系统才会为你关闭文件。假如程序因为错误而异常终止，文件就不会关闭，并可能损坏。程序在结束文件处理后立即关闭文件，文件损坏的概率就大大降低。其二，你可能希望程序将输出发送给一个文件，以后又将那些输出读回程序。为此，程序应该在完成向文件的写入之后立即关闭文件，再用 open 函数将文件连接到一个输入流(也可以打开一个文件，并同时进行输入和输出，但方式稍有区别，我们不准备讨论这种技术)。

类与对象入门

视频讲解：*Objects and File I/O Streams*

无论上一节讨论的 inStream 和 outStream 流，还是预定义的 cin 和 cout 流，它们其实都是对象。**对象**(Object)是一个变量，和它关联的既有函数，也有数据。例如，inStream 和 outStream 流都关联了名为 open 的函数。下面展示了两个函数的示例调用以及 inStream 和 outStream 对象的声明：

```
ifstream inStream;
ofstream outStream;
inStream.open("infile.dat");
outStream.open("outfile.dat");
```

之所以采用这种特殊表示法，是因为和 inStream 对象关联的 open 函数有别于和 outStream 对象关联的 open 函数。一个函数打开文件以进行输入，另一个则打开文件以进行输出。当然，两个函数是相似的，它们都“打开文件”。之所以为两个函数取同一个名字，正是因为它们在直觉上有一些共通之处。然而，名为 open 的两个函数仍然是不同的函数，即使它们只有少许不同。编译器遇到对一个名为 open 的函数的调用时，必须判断所指的函数是哪一个。编译器通过查看圆点之前的对象名称来判断所指的函数。本例中，对象名称可能是 inStream 或 outStream。与对象关联的函数称为**成员函数**(member function)。例如，open 是 inStream 对象的成员函数，另一个 open 是 outStream 对象的成员函数。

如前所述，不同对象可能含有不同的成员函数。这些函数可能同名(比如两个 open)，也可能具有完全不同的名称。对象的类型决定了对象有哪些成员函数。如果两个对象属于同一个类型，它们可能有不同的值，但是肯定有相同的成员函数。例如，假设声明了以下流对象：

```
ifstream inStream, inStream2;
ofstream outStream, outStream2;
```

那么 inStream.open 和 inStream2.open 函数是同一个函数。类似地，outStream.open 和 outStream2.open 函数也是同一个函数(但它们有别于 inStream.open 和 inStream2.open 函数)。

如果一个类型的变量是对象(比如 ifstream 和 ofstream)，这个类型就称为**类**(class)。由于对象的成员函数完全由它的类(也就是它的类型)决定，所以这些函数称为“类的成员

函数”(也称为“对象的成员”)。例如，ifstream 类有一个名为 open 的成员函数，ofstream 类也有一个名为 open 的成员函数。ofstream 类还有一个名为 precision 的成员函数，但 ifstream 类没有名为 precision 的成员函数。本书前面已用过 cout 的成员函数 precision，稍后还会详细讨论。

类 和 对 象

对象是关联了函数的变量。这些函数称为**成员函数**。**类**是一种类型，它的变量就是对象。对象所属的类(即对象的类型)决定了对象有哪些成员函数。

在程序中调用成员函数时，总是要指定一个对象(通常在函数名前附加对象名和一个圆点)。如下例所示：

```
inStream.open("infile.dat");
```

之所以要指定对象名，一个原因是函数可能以某种方式影响对象。在前面的例子中，open 函数调用将 infile.dat 文件连接到 inStream 流，所以它需要知道这个流名称(对象名)。

在以下函数调用中：

```
inStream.open("infile.dat");
```

圆点称为**圆点操作符**(dot operator)，圆点前的对象称为**调用对象**(calling object)或者发出调用的对象。从某个角度看，调用对象好比函数的一个额外实参(函数可将调用对象当作实参进行更改)。但是，调用对象在函数调用中具有更重要的意义。调用对象决定了函数名的含义。编译器根据调用对象的类型判断函数名的含义。以上述 open 函数调用为例，inStream 对象的类型决定了 open 函数的含义。

函数名 close 类似于 open。ifstream 和 ofstream 类都有名为 close 的成员函数，两者都“关闭文件”。但它们以不同方式关闭文件，因为文件以不同方式打开，而且以不同的方式操纵。本章后文将讨论 ifstream 和 ofstream 类更多的成员函数。

调用成员函数

语法　　　　圆点操作符

```
Calling_Object.Member_Function_Name(Argument_List);
```

示例

```
inStream.open("infile.dat");
outStream.open("outfile.dat");
outStream.precision(2);
```

Member Function Name(成员函数名)的含义由 *Calling Object*(调用对象)的类(即类型)决定。

编程提示：检查文件是否成功打开

open 调用可能因许多原因而失败。例如，打开不存在的输入文件，open 调用就会失败。在这种情况下，可能不会报告错误消息，你的程序将在你不知情的情况下执行非预期的操

作。因此，在 open 函数调用后，应该总是进行测试，判断 open 调用是否成功。不成功就终止程序(或采取其他适当行动)。

可用成员函数 fail 测试一个流操作是否失败。ifstream 和 ofstream 类都有名为 fail 的成员函数。fail 函数不取任何参数，返回一个 bool 值。例如，要为 inStream 流调用 fail 函数，可以像下面这样写：

```
inStream.fail()
```

这是一个布尔表达式，可用于控制 while 循环或 if-else 语句。

每个 open 调用之后都应当立即调用 fail。open 调用失败，fail 函数返回 true(也就是满足布尔表达式)。例如，假定以下 open 调用失败：

```
inStream.open("stuff.dat");
if (inStream.fail())
{
    cout << "Input file opening failed.\n";
    exit(1);   <---------------- 终止程序
}
```

程序会输出错误消息并终止。调用成功，fail 函数返回 false，程序得以继续。

fail 是成员函数，所以要用流名称加圆点来调用。当然，inStream.fail 只能判断 inStream.open 调用是否成功。其他任何流作为调用对象，inStream.fail 判断不了它们的 open 调用是否成功。

前面的 exit 语句与类无关，也和流没有直接关系。但这种情况一般都要用到它。exit 语句造成程序立即终止。exit 函数将它的实参返回给操作系统。为了使用 exit 函数，程序中必须包含以下 include 预编译指令：

```
#include <cstdlib>
```

使用 exit 时，程序还必须包含以下预编译指令，它通常要放在文件起始位置，或者放在使用了 exit 的函数体的起始位置：

```
using namespace std;
```

exit 函数是预定义函数，它取一个整数作为实参。根据约定，如果是因为一个错误而调用 exit，就使用 1 作为实参；否则使用 0[①]。在我们的例子中，无论使用哪个整数都没有区别，但最好还是遵守上述约定，因为在高级编程环境中，这一点可能变得重要。 ■

exit 语句

exit 语句的形式如下：

```
exit(Integer_Value);
```

执行 exit 语句，程序立即终止。可用任何 *Integer_Value*(整数值)，但根据约定，如果是因为一个错误而调用 exit，就使用 1，其他情况下则使用 0。exit 语句是对 exit 函数的一个调用，该函数在头文件为 cstdlib 的一个库中。所以，使用 exit 语句的任何程序都必须包含以下指令：

① UNIX 和 Windows 操作系统用 1 表示错误，用 0 表示成功；其他操作系统则可能相反。具体用哪个值，询问你的老师。

```
#include <cstdlib>
using namespace std;
```

(这两个指令不必紧挨在一起。位置安排和以前见过的其他指令相同。)

图 6.2 重写了图 6.1 的程序，引入测试机制来检查输入和输出文件是否成功打开。它采用和图 6.1 的程序完全相同的方式处理文件。具体地说，如 infile.dat 文件存在，而且包含如图 6.1 所示的内容，图 6.2 的程序就会创建如图 6.1 所示的 outfile.dat 文件。但如果出现错误，而且其中一个 open 调用失败，图 6.2 的程序就会终止，并向屏幕发送恰当的错误消息。例如，如果不存在 infile.dat 文件，inStream.open 调用就会失败，程序将终止，并在屏幕上显示错误消息。

注意是用 cout 输出错误消息；这是因为我们希望将错误消息发送到屏幕，而不是发送到文件。由于程序使用 cout 输出到屏幕(同时还要执行文件 I/O)，所以添加了一个#include <iostream>指令。事实上，如果程序已包含#include <fstream>，#include <iostream>就是多余的。但添加也没有什么问题，至少能提醒你程序同时使用了屏幕输出和文件 I/O。

图 6.2　执行文件 I/O 时检查 open 调用是否成功

```
1  //从文件 infile.dat 中读取 3 个数，求和，将结果写入文件 outfile.dat
2
3  #include <fstream>
4  #include <iostream>
5  #include <cstdlib>
6  int main()
7  {
8      using namespace std;
9      ifstream inStream;
10     ofstream outStream;
11     inStream.open("infile.dat");
12     if(inStream.fail())
13     {
14         cout << "Input file opening failed.\n";
15         exit(1);
16     }
17     outStream.open("outfile.dat");
18     if(outStream.fail())
19     {
20         cout << "Output file opening failed.\n";
21         exit(1);
22     }
23     int first, second, third;
24     inStream >>first >>second >> third;
25     outStream << "The sum of the first 3\n"
26               << "numbers in infile.dat\n"
27               << "is " << (first + second + third)
28               << endl;
29     inStream.close();
30     outStream.close();
31     return 0;
32 }
```

屏幕输出(如果 infile.dat 文件不存在)：

```
Input file opening failed.
```

文件 I/O 技术

如前所述，操作符>>和<<可像应用于 cin 和 cout 那样应用于连接到文件的流。但文件 I/O 的编程风格有别于屏幕和键盘 I/O。读取键盘输入应提示并回显用户输入，如下所示：

```
cout << "Enter the number: ";  // 提示输入
cin >> theNumber;
cout << "The number you entered is " << theNumber; // 回显输入
```

但从文件获取输入时，就不要像这样提示或回显输入，因为没人看到(并响应)提示和回显。从文件获取输入时，必须确定文件中的数据确实是程序要求的数据。然后，程序假定数据就绪，直接开始读取输入文件。假设 inFile 是连接到一个输入文件的流变量，将以上 3 行代码替换为下面这一行代码，从而用该文件中的输入替换前面的键盘和屏幕 I/O：

```
inFile >> theNumber;
```

可同时打开多个流进行输入输出。因此，一个程序可以既从键盘获取输入，也从一个或多个文件获取输入。同一个程序可以既向屏幕发送输出，也向一个或多个文件发送输出。另外，还可从键盘获取所有输入，并将输出同时发送到屏幕和文件。输入流和输出流可任意组合。本书大多数例子用 cin 和 cout 执行键盘和屏幕 I/O。但很容易修改这些程序，让程序从文件获取输入，并且/或者将输出发送到文件。

文件 I/O 语句小结

本例假定输入来自 infile.dat 文件，输出发送到 outfile.dat 文件。

- 在程序文件中添加以下 include 预编译指令：

```
#include <fstream>   ←  用于文件 I/O
#include <iostream>  ←  用于 cout
#include <cstdlib>   ←  用于 exit
```

- 为输入流选择流名称(例如 inStream)，声明为 ifstream 类型的变量；为输出文件选择流名称(例如 outStream)，声明为 ofstream 类型的变量。

```
using namespace std;
ifstream inStream;
   ofstream outStream;
```

- 将外部文件名作为实参，调用 open 函数将每个流都连接到文件。记住用成员函数 fail 测试 open 调用是否成功。

```
inStream.open("infile.dat");
if (inStream.fail())
  {
    cout  << "Input file opening failed.\n";
    exit(1);
 }
outStream.open("outfile.dat");
```

```
    if (outStream.fail())
    {
        cout << "Output file opening failed.\n";
        exit(1);
    }
```

- 就像使用 cin 从键盘获取输入一样，使用 inStream 流从 infile.dat 文件获取输入，如下所示：

```
inStream >> someVariable >> someOtherVariable;
```

- 就像使用 cout 将输出发送到屏幕一样，使用 outStream 流将输出发送到 outfile.dat 文件，如下所示：

```
outStream << "someVariable = "
          << someVariable << endl;
```

- 使用 close 函数关闭流。

```
inStream.close();
outStream.close();
```

√ 自测题

1. 假设程序中使用了一个名为 fin 的流，连接到一个输入文件；还使用了一个名为 fout 的流，连接到一个输出文件。怎样声明 fin 和 fout？需在程序文件中添加什么 include 预编译指令？
2. 继续写上一题的程序，并希望程序从 stuff1.dat 文件获取输入，将输出发送到 stuff2.dat 文件。为了将 fin 流连接到 stuff1.dat 文件，将 fout 流连接到 stuff2.dat 文件，需在程序中添加什么语句？必须设计正确的检查机制，确保文件成功打开。
3. 继续写前两题的程序，现已结束文件处理，不再需要从 stuff1.dat 文件获取输入，也不再需要将输出发送到 stuff2.dat 文件。怎样关闭这些文件？
4. 假定要更改图 6.1 的程序，使其将输出发送到屏幕，而不是发送到 outfile.dat 文件(输入仍从文件 infile.dat 中获得)，应怎样修改？
5. 程序要用 exit 函数需包含什么预编译指令？
6. 继续上一题，exit(1)的实参有什么作用？
7. 假设 bla 是一个对象，dobedo 是它的成员函数，获取一个 int 类型的实参。怎样写一个调用，用实参 7 来调用 bla 对象的 dobedo 成员函数。
8. 文件和程序的普通变量有哪些共同点？不同点又在哪里？
9. 列举与 iostream 关联的成员函数，至少要列举 3 个，举例说明每个成员函数的用法。
10. 假定程序已读取了文件的一半内容。为了重读第一行，程序必须对文件采取什么操作？
11. 前面说过，一个文件有两个名称。这两个名称是什么？分别在什么时候使用？

追加到文件(选读)

将输出发送到文件时，必须先用成员函数 open 打开文件，再将文件连接到 ofstream 类型的流。以前的例子采取的方式(用单独一个实参指定文件名)肯定会生成一个空文件。如指定的文件存在，它以前的内容就会丢失。还可采用另一种方式打开文件，使程序输出追加到文件中现有数据之后。

用带两个参数的 open 将输出追加到 important.txt 文件：

```
ofstream outStream;
outStream.open("important.txt", ios::app);
```

文件 important.txt 不存在就新建一个具有该名称的空文件。如存在，程序的所有输出都会追加到文件尾，文件原有数据不会丢失。图 6.3 进行了演示。

图 6.3　在文件尾追加数据

```
1  // 将数据追加到 data.txt 文件末尾
2  #include <fstream>
3  #include <iostream>
4
5  int main()
6  {
7       using namespace std;
8
9      cout << "Opening data.txt for appending.\n";
10     ofstream fout;
11     fout.open("data.txt", ios::app);
12     if (fout.fail())
13     {
14         cout << "Input file opening failed.\n";
15         exit(1);
16     }
17
18     fout << "5 6 pick up sticks.\n"
19          << "7 8 ain't C++ great!\n";
20
21     fout.close();
22     cout << "End of appending to file.\n";
23
24    return 0;
25 }
```

示范对话

data.txt (程序运行前)	**data.txt** (程序运行后)
1 2 buckle my shoe. 3 4 shut the door.	1 2 buckle my shoe. 3 4 shut the door. 5 6 pick up sticks. 7 8 ain't C++ great!

屏幕输出

```
Opening data.txt for appending.
End of appending to file.
```

第二个实参 ios::app 是特殊常量，它在 iostream 中定义，所以需要以下 include 预编译指令：

```
#include <iostream>
```

程序还应包含以下 using 语句，该语句要么放在文件起始处，要么放在使用了 ios::app 的函数体的起始处：

```
using namespace std;
```

追加到文件尾

要将数据追加到文件尾，使其位于文件原有内容之后，应该像下面这样打开文件。

语法

outputStream.open(FileName, ios::app);

示例
```
ofstream outStream;
outStream.open("important.txt", ios::app);
```

文件名作为输入(选读)

目前为止的程序一直都是直接使用输入/输出文件的文件名。换言之，总是在 open 函数调用中嵌入文件名作为实参，如下例所示：

```
inStream open("infile.dat");
```

但这样有时会不方便。例如，图 6.2 的程序从 infile.dat 文件读取数字，并累加这些数字，结果输出到 outfile.dat 文件。要对名为 infile2.dat 的另一个文件中的数字执行同样的计算，并将累加结果写入名为 outfile2.dat 的另一个文件，就必须在两个 open 调用中更改文件名，并重新编译程序。一个更好的办法要求用户提供输入和输出文件名。这样程序就能在每次运行时使用不同的文件。

文件名是字符串，将在第 8 章详细讨论字符串处理。但在此之前，通过简单的学习，很容易就能掌握字符串的常识，让程序接收文件名作为输入。**字符串**(string)是一个字符序列，我们已在输出语句中用过字符串值，如下所示：

```
cout << "This is a string.";
```

还曾将字符串值作为实参传给成员函数 open。字符串字面值[①](如上述 cout 语句所示)必须用双引号封闭。

程序读入文件名必须有一个能保存字符串值的变量。第 8 章将讨论字符串的细节，目前只需知道如何存储文件名。下例展示了如何声明用于容纳字符串值的变量：

```
char fileName[16];
```

这和以前的 char 变量声明差不多，只是变量名后添加了一个方括号中的整数，它指定变量存储的字符串的最大字符数。该数字必须比字符串的实际最大字符数大 1。所以在上例中，fileName 变量最多能包含 15 个字符。可将名称 fileName 替换为其他标识符(只要不是 C++ 关键字)。另外，数字 16 也可替换为其他正整数。

可像输入其他类型的值那样，将字符串值输入字符串变量。以下面的代码为例：

```
cout << "Enter the file name (maximum of 15 characters):\n";
cin >> fileName;
cout << "OK, I will edit the file " << fileName << endl;
```

上述代码的一个可能的对话如下：

```
Enter the file name (maximum of 15 characters):
myfile.dat
OK, I will edit the file myfile.dat
```

一旦将文件名输入字符串变量(比如 fileName)，就可将该字符串变量用作成员函数 open 的实参。例如，以下代码将输入文件流 inStream 连接到由 fileName 变量指定的文

① “字面值”原文是 literal，有多种译法，但没有一种占绝对优势。其他译法还有“文本常量”和“直接量”。——译注

件(并用成员函数 fail 核实文件是否成功打开):

```
ifstream inStream;
inStream.open(fileName);
if (inStream.fail())
{
    cout << "Input file opening failed.\n";
    exit(1);
}
```

注意，字符串变量作为成员函数 open 的实参使用时不添加任何引号。

图 6.4 重写了图 6.2 的程序，使其能从用户指定的文件读取输入，并将输出发送到用户指定的文件。输入和输出文件的名称分别读入字符串变量 inFileName 和 outFileName。然后，两个字符串变量作为实参传给成员函数 open。注意两个字符串变量的声明。在每个字符串变量名称之后，都必须用方括号来包含一个数字。

字符串变量不是普通变量，不能完全像使用普通变量那样使用它，尤其不能用赋值语句更改字符串变量的值。

图 6.4 输入一个文件名(选读)

```
1  // 从用户指定文件读取 3 个数，求和，结果写入用户指定的另一个文件
2
3  #include <fstream>
4  #include <iostream>
5  #include <cstdlib>
6
7  int main()
8  {
9      using namespace std;
10     char inFileName[16], outFileName[16];
11     ifstream inStream;
12     ofstream outStream;
13
14     cout << "I will sum three numbers taken from an input\n"
15          << "file and write the sum to an output file.\n";
16     cout << "Enter the input file name (maximum of 15 characters):\n";
17     cin >> inFileName;
18     cout << "Enter the output file name (maximum of 15 characters):\n";
19     cin >> outFileName;
20     cout << "I will read numbers from the file "
21          << inFileName << " and\n"
22          << "place the sum in the file "
23          << outFileName << endl;
24
25     inStream.open(inFileName);
26     if (inStream.fail())
27     {
28         cout << "Input file opening failed.\n";
29         exit(1);
30     }
31
32     outStream.open(outFileName);
33     if (outStream.fail())
34     {
35         cout << "Output file opening failed.\n";
36         exit(1);
37     }
38     int first, second, third;
39     inStream >> first >> second >> third;
40     outStream << "The sum of the first 3\n"
```

```
41                  << "numbers in " << inFileName << endl
42                  << "is " << (first + second + third)
43                  << endl;
44
45      inStream.close();
46      outStream.close();
47
48      cout << "End of Program.\n";
49      return 0;
50  }
```

numbers.dat (不被程序改变)	sum.dat (程序运行后)
1 2 3 4	The sum of the first 3 numbers in numbers.dat is 6

示范对话

```
I will sum three numbers taken from an input
file and write the sum to an output file.
Enter the input file name (maximum of 15 characters):
numbers.dat
Enter the output file name (maximum of 15 characters):
sum.dat
I will read numbers from the file numbers.dat and
place the sum in the file sum.dat
End of Program.
```

6.2　流 I/O 工具

我相信，当你看到这些诗印在精美的四开纸上的时候，你会喜欢的，印在纸上的诗行一如清澈的溪流顺着沃野的岸缘蜿蜒地潺潺流过。

——*谢立丹，《造谣学校》*

用流函数格式化输出

对程序输出的布局进行调整称为对输出进行**格式化**。在 C++中，可以用一些命令来控制格式。这些命令可指定大量格式细节，比如项与项之间的空格数，小数点后保留多少位等。以前采用常规方式(而不是 e 记数法)输出美元金额时，为了保留两位小数，曾用过三个输出格式化指令。输出货币金额前，在程序中插入了以下“魔法配方”：

```
cout.setf(ios::fixed);
cout.setf(ios::showpoint);
cout.precision(2);
```

学习了流的对象表示法之后，就可真正理解该“魔法配方”和其他格式化命令。

首先要注意，可为任何输出流使用这些格式化命令。例如，当程序将输出发送给与输出流 outStream 连接的文件时，可用同样的命令确保向文件中写入货币格式的数字：

```
outStream.setf(ios::fixed);
outStream.setf(ios::showpoint);
outStream.precision(2);
```

为理解该“魔法配方”，让我们按相反顺序解释上述每一条指令。

每个输出流都有名为 precision 的成员函数。程序一旦为 outStream 流执行了上述 precision 调用，从此以后，向那个流输出带小数点的任何数字时，要么总共保留两个有效数字，要么在小数点之后保留两位——具体由你使用的编译器来决定。例如，假定编译器决定保留两个有效数字，那么以下输出都是可能的：

```
23.          2.2e7     2.2       6.9e-1        0.00069
```

相反，假定编译器决定在小数点之后保留两位，那么以下输出都是可能的：

```
23.56        2.26e7    2.21      0.69          0.69e-4
```

本书假定编译器在小数点后保留两位。

precision 函数调用只对调用中指定的流生效。如果还有一个名为 outStreamTwo 的输出流，outStream.precision 调用只影响向 outStream 流的输出，不影响向 outStreamTwo 流的输出。当然，也可为 outStreamTwo 调用 precision，甚至可为输出到 outStreamTwo 的数字指定不同小数位数，如下所示：

```
outStreamTwo.precision(3);
```

“魔法配方”中的其他指令比成员函数 precision 稍微复杂一些。下面是对成员函数 setf 的两个调用，调用对象(发出调用的对象)是 outStream：

```
outStream.setf(ios::fixed);
outStream.setf(ios::showpoint);
```

setf 是 set flags(设置标志)的缩写。**标志(flag)**是以二选一的方式来做一件事情的指令。标志作为 setf 的实参，该标志就会指示计算机以特定方式将输出写入那个流。不同标志对流有不同的指示。

上例展示了两个 setf 函数调用，分别设置了 ios::fixed 标志和 ios::showpoint 标志。其中，ios::fixed 标志导致流采用**定点记数法(fixed-point notation)**输出 double 类型的数字；这其实就是平时书写数字时采用的方式，只是换了一种说法。一旦设置 ios::fixed 标志(通过调用 setf)，输出到那个流的所有浮点数(比如 double 类型的数字)就会采用日常生活中的常规方式来写入，而不是采用 e 记数法。

ios::showpoint 标志要求流总是在浮点数(比如 double 类型的数字)中包含小数点。所以，如果准备输出的是 2.0，输出的就肯定是 2.0，不会是 2，换言之，即使小数点之后的所有数字都是 0，输出中始终都会包含小数点。图 6.5 总结了部分常用标志，并对它们的含义进行了解释。

另一个有用的标志是 ios::showpos。为流设置该标志，就会为输出到那个流的正数添加正号。要在正数前显示正号，请在程序中插入下面这行代码：

```
cout.setf(ios::showpos);
```

注意，即使不设置任何标志，负数之前也会显示负号。

一个常用的格式化函数是 width。例如，可以像下面这样为 cout 流调用 width：

```
cout << "Start Now";
cout.width(4);
cout << 7 << endl;
```

这段代码导致屏幕上显示以下输出：

```
Start Now   7
```

在输出中，字母 w 与数字 7 之间有 3 个空格。width 函数告诉流一个输出项需要占多少个字符位置(即域宽)。本例中，输出项(也就是数字 7)只占一个字符位置，而 width 要求使用 4 个字符位置，所以其他 3 个位置用空格填充。如输出所需的字符位置数目超过了在 width 函数调用中指定的数目，就自动补足缺少的字符位置。总之，输出项始终都会完整输出，不会被截短，无论为 width 指定的参数是什么。

图 6.5　setf 的格式化标志

标志	含义	默认
ios::fixed	如果设置这个标志，就不用 e 记数法写浮点数(设置该标志会自动取消设置 ios::scientific 标志)	未设置
ios::scientific	如果设置这个标志，会用 e 记数法写浮点数(设置该标志会自动取消设置 ios::fixed 标志) 如果既没有设置 ios::fixed，也没有设置 ios::scientific，就由系统决定如何输出每个数字	未设置
ios::showpoint	如果设置这个标志，就始终为浮点数显示小数点和尾随的 0。如果没有设置这个标志，而且一个数字在小数点之后全是 0，那么当这个数字输出时，就可能不会显示小数点和尾随的 0	未设置
ios::showpos	如果设置这个标志，正整数之前会输出一个正号	未设置
ios::right	如果设置这个标志，同时调用成员函数 width 指定了域宽，输出的下一项会对齐指定域的右侧(右对齐)。也就是说，在输出的项之前，会根据需要添加填充空格(设置该标志会自动取消设置 ios::left 标志)	已设置
ios::left	如果设置这个标志，同时调用成员函数 width 指定域宽，输出的下一项会对齐指定域的左侧(左对齐)。也就是说，在输出的项之后，根据需要添加填充空格(设置该标志会自动取消设置 ios::right 标志)	未设置

对 width 函数的调用只适用于下一个要输出的项。要输出 12 个数字，而且每个都占 4 个字符位置，就必须调用 12 次 width。如果觉得这样很麻烦，可以使用流操纵元 setw，详情参见下一节。

设置的任何标志都可用 unsetf 函数取消。例如，以下语句导致程序停止为输出到 cout 流的正整数显示正号：

```
cout.unsetf(ios::showpos);
```

“标志”术语释疑

为什么将 setf 的参数(比如 ios::showpoint)称为标志？ios::这个奇怪的表示法究竟有何含义？

标志表示某些可以开关的东西。[①]但人们之所以记不住这个词的准确来源，是因为程序员现在一般都说：“如标志被设置，则做某事”。显然，“开”和“设置”还是有一定差距的。任何情况下，如 ios::showpoint 标志被设置(也就是说，作为 setf 的实参)，调用 setf 函数的那个流就会具有图 6.5 为该标志总结的行为。同样地，其他任何标志被设置(也就是说，作为 setf 的实参)，流就会具有图 6.5 为该标志总结的行为。

至于 ios::这种奇怪的表示法，ios 是“输入或输出流”(input or output stream)的简称，::的意思则是说：“::之后术语的具体含义以::之前的术语为背景。”本书以后还会进一步讨论::表示法。

操纵元

操纵元(manipulator)是以非传统方式调用的函数。调用操纵元后，它本身又会调用一个成员函数。操纵元位于插入操作符<<之后，好似它本身就是下一个要输出的项。和传统函数一样，操纵元可以有、也可以没有参数。以前其实已见过一个操纵元，即 endl。本节讨论两个新的操纵元：setw 和 setprecision。

操纵元 setw 和成员函数 width(参见前文描述)所做的事情完全一样。要调用 setw 操纵元，在插入操作符<<之后写上 setw 即可。这如同将该操纵元发送到输出流，后者随即调用成员函数 width。例如，以下语句用不同域宽输出数字 10，20 和 30：

```
cout << "Start" << setw(4) << 10
     << setw(4) << 20 << setw(6) << 30;
```

输出如下所示(注意 10 之前有 2 个空格，20 之前有 2 个空格，30 之前有 4 个空格)：

```
Start  10  20    30
```

流操纵元 setprecision 所做的事情和成员函数 precision(参见前文描述)完全一样。但 setprecision 调用要放在插入操作符<<之后，这和调用 setw 操纵元是类似的。例如，以下语句为指定的数字输出小数部分，小数位数由 setprecision 调用来指定：

```
cout.setf(ios::fixed);
cout.setf(ios::showpoint);
cout << "$" << setprecision(2) << 10.3 << endl
     << "$" << 20.5 << endl;
```

以上语句的输出如下：

```
$10.30
$20.50
```

使用 setprecision 操纵元设置小数位数时，和成员函数 precision 的情况一样，设

① flag 最早被翻译成“旗标”，那其实更准确。用英语解释 flag 会更容易理解。When the flag is up, do it；when the flag is down, don't do it(旗子升起来，做某事；旗子放下去，就不要做)。翻译成“标志”失去了二选一的意味。——译注

置会一直生效，直到把它重设为其他数字(通过再次调用 setprecision 或 precision)。

要使用 setw 或 setprecision 操纵元，必须在程序中包含以下预编译指令：

```
#include <iomanip>
```

与此同时，程序中还应该包含以下语句：

```
using namespace std;
```

√ 自测题

12. 以下语句输出什么(假定嵌入完整而正确的程序中，而且包含了正确的 include 预编译指令)？

```
cout << "*";
cout.width(5);
cout << 123
     << "*" << 123 << "*" << endl;
cout << "*" << setw(5) << 123
     << "*" << 123 << "*" << endl;
```

13. 以下语句输出什么(假定嵌入完整而正确的程序中，而且包含了正确的 include 预编译指令)？

```
cout << "*" << setw(5) << 123;
cout.setf(ios::left);
cout << "*" << setw(5) << 123;
cout.setf(ios::right);
cout << "*" << setw(5) << 123 << "*" << endl;
```

14. 以下语句输出什么(假定嵌入完整而正确的程序中，而且包含了正确的 include 预编译指令)？

```
cout << "*" << setw(5) << 123 << "*"
     << 123 << "*" << endl;
cout.setf(ios::showpos);
cout << "*" << setw(5) << 123 << "*"
     << 123 << "*" << endl;
cout.unsetf(ios::showpos);
cout.setf(ios::left);
cout << "*" << setw(5) << 123 << "*"
     << setw(5) << 123 << "*" << endl;
```

15. 执行以下语句会向文件 stuff.dat 发送什么输出(假定嵌入完整而正确的程序中，而且包含了正确的 include 预编译指令)？

```
ofstream fout;
fout.open("stuff.dat");
fout << "*" << setw(5) << 123 << "*"
     << 123 << "*" << endl;
fout.setf(ios::showpos);
fout << "*" << setw(5) << 123 << "*"
     << 123 << "*" << endl;
fout.unsetf(ios::showpos);
fout.setf(ios::left);
fout << "*" << setw(5) << 123 << "*"
     << setw(5) << 123 << "*" << endl;
```

16. 以下语句输出什么(假定嵌入完整而正确的程序中，而且包含了正确的 include 预编译指令)？

```
cout << "*" << setw(3) << 12345 << "*" << endl;
```

17. 对输出进行格式化时，以下标志常量可与 stream 的成员函数 setf 配合使用。它们分别有何作用？

    a. ios::fixed.
    b. ios::scientific
    c. ios::showpoint
    d. ios::showpos
    e. ios::right
    f. ios::left

18. 以下代码从 infile.dat 读取输入，将输出发送给 outfile.dat。要将输出发送到屏幕，需对代码进行什么改动(输入不变，仍来自 infile.dat)？

```
// 在不同文件之间复制三个 int 数字
#include <fstream>
int main()
{
    using namespace std;

    ifstream instream;
    ofstream outstream;

    instream.open("infile.dat");
    outstream.open("outfile.dat");
    int first, second, third;
    instream >> first >> second >> third;
    outstream << "The sum of the first 3" << endl;
              << "numbers in infile.dat is " << endl
              << (first + second + third) << endl;
    instream.close();
    outstream.close();
    return 0;
}
```

流作为函数实参

流可作为函数实参使用。唯一限制就是形参必须传引用。流参数不能传值。例如，图 6.6 的 makeNeat 函数有两个流参数：一个是 ifstream 类型，是连接到一个输入文件的流；另一个是 ofstream 类型，是连接到一个输出文件的流。后续两个小节将讨论这个程序的其他特点。

图 6.6　格式化输出

```
1  // 演示输出格式化命令
2  // 读取 rawdata.dat 文件中的所有数字，然后采用美观的格式，
3  // 将数字写到屏幕，同时写到 neat.dat 文件
4  #include <iostream>
5  #include <fstream>
6  #include <cstdlib>
7  #include <iomanip>   <------------ 这是 setw 需要的
8  using namespace std;                              流参数必须传引用
9  void makeNeat(ifstream& messyFile, ofstream& neatFile,
10               int numberAfterDecimalpoint, int fieldWidth);
11 // 前条件：messyFile 和 neatFile 这两个流已用 open 函数连接到文件
12 // 后条件：与 messyFile 流连接的那个文件中的数字写到屏幕，
13 // 同时写到与 neatFile 流连接的那个文件
14 // 每个数字单独占一行，并采用定点记数法(换言之，不采用 e 记数法)，
15 // 而且在小数点之后保留 numberAfterDecimalpoint 位小数；
16 // 每个数字的前面要么添加一个正号，要么添加一个负号，
17 // 而且每个数字都占用宽度为 fieldWidth 的一个域(该函数不关闭文件)
18
19 int main()
20 {
21     ifstream fin;
22     ofstream fout;
23
24     fin.open("rawdata.dat");
25     if (fin.fail())
26     {
27         cout << "Input file opening failed.\n";
28         exit(1);
29     }
30     fout.open("neat.dat");
31     if (fout.fail())
32     {
33         cout << "Output file opening failed.\n";
34         exit(1);
35     }
```

```
36
37      makeNeat(fin, fout, 5, 12);
38
39      fin.close();
40      fout.close();
41
42      cout << "End of program.\n";
43      return 0;
44  }
45
46  // 使用 iostream,fstream 和 iomanip:
47  void makeNeat(ifstream& messyFile, ofstream& neatFile,
48                int numberAfterDecimalpoint, int fieldWidth)
49  {
50      neatFile.setf(ios::fixed);          ← 不采用 e 记数法
51      neatFile.setf(ios::showpoint);      ← 显示小数点
52      neatFile.setf(ios::showpos);        ← 显示正号(+)
53      neatFile.precision(numberAfterDecimalpoint);
54      cout.setf(ios::fixed);
55      cout.setf(ios::showpoint);
56      cout.setf(ios::showpos);
57      cout.precision(numberAfterDecimalpoint);
58
59      double next;
60      while (messyFile >> next)   ← 该条件在还有可读的下个数字时满足
61      {
62          cout << setw(fieldWidth) << next << endl;
63          neatFile << setw(fieldWidth) << next << endl;
64      }
65  }
```

rawdata.dat
(不会被程序更改)

```
10.37      -9.89897
2.313    -8.950  15.0

7.33333   92.8765
-1.237568432e2
```

neat.dat
(程序运行之后)

```
 +10.37000
  -9.89897
  +2.31300
  -8.95000
 +15.00000
  +7.33333
 +92.87650
-123.75684
```

屏幕输出

```
 +10.37000
  -9.89897
  +2.31300
  -8.95000
 +15.00000
  +7.33333
 +92.87650
-123.75684
End of program.
```

编程提示：检查文件尾

都在这里了，没有更多了。

——埃塞尔·巴里摩尔 (1879—1959)

经常需要读取文件中的所有数据。例如，你可能希望求文件中所有数字的平均值。由于每次执行程序都可能从不同文件读取输入，所以程序事先不知道文件中含有多少数字。这种情况下，可要求程序从文件中连续读取数字，直到没有更多数字可供读取为止。假定 inStream 是连接到输入文件的流，可以这样叙述计算平均值的算法：

```
double next, sum = 0;
int count = 0;
while (还有可供读取的数字)
{
    inStream >> next;
    sum = sum + next;
    count++;
}
平均值是 sum / count
```

根据上述代码，C++代码其实已呼之欲出。但是，仍需用 C++表示以下测试：

(*还有可供读取的数字*)

虽然表面上不合逻辑，但确实能用以下形式表示该测试：

```
(inStream >> next)
```

因此，上述算法可重写为以下 C++代码(伪代码最后一行需进一步处理)：

```
double next, sum = 0;
int count = 0;
while (inStream >> next)
{
    sum = sum + next;
    count++;
}
平均值是 sum / count
```

注意，循环主体稍微有别于伪代码。由于 inStream >> next 已包含到布尔表达式中，所以把它从循环主体中删除了。

以上循环看起来有点儿特别，因为 inStream >> next 既能用于从 inStream 流读取一个数字，也能控制 while 循环的布尔表达式。涉及提取操作符>>的表达式既是行动，也是布尔条件。[①]它既是从输入流获取一个输入数字的指令，也是一个要么满足、要么不满足的布尔表达式。如果还有下个数字可读，就读取那个数字。同时，由于布尔表达式满足，所以再次执行循环主体。如果没有可供读取的数字，就不读取任何数字，布尔表达式也不满足，所以终止循环。本例输入变量 next 的类型为 double，但其他类型(比如 int 和 char)也可以用这种方式检查文件尾。

命名空间的问题

以前尽量让 using 预编译指令局部于函数定义。这是很正确的方案，但现在有个新问题，即函数参数类型在一个命名空间中。图 6.6 的例子需要命名空间 std 中的流类型名称。所以，需要在函数体外部使用 using 预编译指令(或采取其他方案)，使 C++能理解参数的类型名称(比如 ifstream)。为此，最简单的解决方案就是在文件起始处(include 预编译指令之后)添加 using 预编译指令。这正是图 6.6 采取的做法(第 8 行)。

对于这个问题，在文件起始处添加 using 预编译指令是最容易的解决方案，但很多专家认为它并非最佳方案，因为一旦采用这种方案，包含了相同名称的两个命名空间就会发生冲突，而之所以设计命名空间，就是为了在使用不同命名空间中的相同名称。就目前来说，我们只用到了命名空间 std[②]，所以问题并不大。第 12 章要介绍如何用另一种方案解决围绕参数和命名空间所产生的问题。那种方案允许任意使用多个命名空间。

许多程序员喜欢在程序文件起始处添加 using 预编译指令。以下面的 using 预编译指令为例：

```
using namespace std;
```

---

① 从技术上说，布尔条件的工作方式是这样的：对于表示输入流的类，操作符>>的重载由类的函数完成。函数名称就是 operator>>，该操作符函数的返回值是一个输入流引用(istream&或 ifstream&)。一个函数能将流引用自动转换成布尔值。如果流能提取数据，就返回 true；否则返回 false。

② 实际使用了两个命名空间，一个是 std 命名空间，另一个是全局命名空间。后者包含了不在其他命名空间中的所有名称。但目前不必过多关心这个技术细节。

本书许多程序都没有将这个 using 预编译指令放在程序文件起始处，而是将它放在需要命名空间 std 的每个函数定义的起始处(紧跟在起始花括号之后)，例如图 6.3，图 5.11 的例子则更具代表性。本书所有程序，无论是已经出现还是将要出现的，都可以像图 6.6 那样，在整个程序中只在文件起始处为命名空间 std 添加 using 指令，并让那个 using 指令紧跟在 include 指令之后。对于命名空间 std，在几乎所有情况下，它的 using 指令都能安全地放在文件起始处。但对于另一些命名空间，一个 using 指令可能不够用。只是在短时间内，你不会遇到这种情况。

我们主张将 using 指令放在函数定义(或其他较小的代码单元)的内部，避免与其他任何 using 预编译指令产生冲突。这样可养成正确使用命名空间的好习惯，为将来从事编程工作，编写复杂程序打下好的基础。但与此同时，假如为了遵循这一规则，会使当前讨论的问题复杂化，我们偶尔也会违反这一规则。如果你正在上课，就按照老师的要求去做。否则，请自行决定 using 指令的位置。

编程实例　整理文件格式

图 6.6 的程序从 rawdata.dat 文件获取输入，并用美观的格式将输出同时发送到屏幕和 neat.dat 文件。程序将数字从 rawdata.dat 复制到 neat.dat，同时用格式化指令来写入。每个数字都单独占一行，并占用宽度为 12 的一个域，换言之，每个数字的前面都会自动添加空格，确保空格和数字总共占用 12 个字符位置。数字采用常规记数法来写入(不用 e 记数法)。每个数字都显示 5 位小数，而且每个数字都有正号或负号。发送到屏幕和 neat.dat 文件的输出几乎一样，只是屏幕输出多了一行(宣布程序终止)。程序使用了一个名为 makeNeat 的函数，参数包括一个输入文件流和一个输出文件流。

√ 自测题

19. 执行以下代码，会产生什么输出？假定 list.dat 文件已包含如后所述的数据，并假定所有代码都嵌入一个完整而正确的程序中，且已包含正确的 include 预编译指令。

```
ifstream ins;
ins.open("list.dat");
int count = 0, next;
while (ins >> next)
{
    count++;
    cout << next << endl;
}
ins.close();
cout << count;
```

list.dat 文件包含以下 3 个数字(而且只包含这些数字)：

```
1     2
3
```

20. 写 void toScreen 函数定义。函数有一个 ifstream 类型的形参 fileStream。函数前后条件如下：

```
// 前条件：调用成员函数 open 将 fileStream 流连接到一个文件
// 文件只包含一个整数列表
// 后条件：与 fileStream 流连接的文件中的数字写到屏幕(每行一个)，该函数不关闭文件
```

21. (要求已学过本章 6.1.7 节)给定以下 string 变量声明和输入语句：

```
#include <iostream>
using namespace std;
// ...
char name[21];
cout >> name;
```

假定上述代码已嵌入一个正确的程序。在字符串变量 name 中，最多能输入一个多长的名称？

6.3　字符 I/O

普娄尼阿斯：“你读什么呢？殿下。”
哈姆雷特：“字，字，字。”

——*威廉·莎士比亚，《哈姆雷特》*

所有数据都作为字符数据输入和输出。程序输出数字 10 时，实际输出两个字符：'1'和'0'。类似地，输入数字 10 时，先后输入的是字符'1'和'0'。计算机将这个 10 解释成两个字符，还是解释成一个数字，要取决于你的程序怎样写。但不管怎样写，计算机硬件读取的始终是'1'和'0'，而不是数字 10。字符和数字之间的这种转换通常自动进行，所以不需要关心细节。但有些时候，这种自动转换反而碍事。因此，C++为字符数据的输入和输出提供了一些低级的辅助机制。这些低级机制不支持自动转换。这样就可绕过自动转换机制，完全按自己的意愿执行输入和输出。如果愿意，甚至可以自己写输入和输出函数，用罗马记数法读写数字。

get 和 put 成员函数

get 函数允许读取输入的一个字符，并把它存储到 char 类型变量。每个输入流都提供了 get 成员函数，无论是输入文件流，还是 cin 流。虽然后文要将 get 描述成 cin 流的一个成员函数，但将它用于输入文件流时，工作方式和用于 cin 时完全相同。所以，可为输入文件流照搬为 get 所讲的一切。

以前曾为 cin 使用提取操作符>>，目的是读取字符输入(或其他任何输入；记住，所有数据都作为字符输入和输出)。以前使用提取操作符>>时，有的工作是自动完成的，比如忽略空白。相反，使用成员函数 get，一切都不会自动发生。例如，如果希望使用 cin.get 时忽略空白，必须手动编写代码，让它读取并丢弃空白字符。

成员函数 get 获取 char 类型的变量作为参数。该参数将接收从输入流读取的输入字符。例如，以下语句从键盘读取下一个输入字符，并将其存储到变量 nextSymbol 中：

```
char nextSymbol;
cin.get(nextSymbol);
```

程序以这种方式能读取任何字符。如下个输入字符是空白字符，上述代码不会忽略它，而是读取该空白字符，并将 nextSymbol 的值设为该空白字符。如果下个输入字符是换行符'\n'；也就是说，程序已达输入行末尾，上述 cin.get 调用会将 nextSymbol 的值设为'\n'。

虽然写成两个符号，但换行符'\n'在 C++中只是一个字符。和成员函数 get 一起使用

时，换行符'\n'能像其他任何字符一样输入和输出。例如，如果程序包含以下代码：

```
char c1, c2, c3;
cin.get(c1);
cin.get(c2);
cin.get(c3);
```

并提供了以下两行输入：

```
AB
CD
```

也就是说，输入 AB 后按 Enter 键，再输入 CD 并按 Enter 键。结果不出所料，c1 的值设为'A'，c2 的值设为'B'。但在填充变量 c3 时，由于使用的是成员函数 get，所以情况有别于使用提取操作符>>。执行上述代码，c3 的值会被设为'\n'；也就是说。c3 的值被设为换行符而不是'C'。

'\n' 和 "\n"

'\n'和"\n"表面上区别不大。在 cout 语句中效果一样。但任何情况下两者都不能互换。'\n'是 char 类型的值，可存储到 char 类型的变量中。"\n"是字符串，凑巧只有一个字符。所以，"\n"不属于 char 类型，不能存储到 char 类型的变量中。

可用成员函数 get 检测行尾。以下循环读取一行输入，并在传入换行符'\n'后终止。然后，后续所有输入都将从下一行的开头读取。由于这是我们的第一个例子，所以只是简单地回显了输入。但是，还可利用同样的技术对输入进行其他任何处理：

```
cout << "Enter a line of input and I will echo it:\n";
char symbol;
do
{
    cin.get(symbol);
    cout << symbol;
} while (symbol != 'n');
cout << "That's all for this demonstration.";
```

该循环将读取并回显任何一个输入行，其中包括空白字符。代码的一个示范对话如下：

```
Enter a line of input and I will echo it:
Do Be Do 1 2     34
Do Be Do 1 2     34
That's all for this demonstration.
```

注意，换行符'\n'同时被读取和输出。由于它被输出，所以以"That's"开头的那个字符串会另起一行显示。

成员函数 get

每个输入流都有名为 get 的成员函数，它读取一个输入字符。和提取操作符不同，无论下个输入字符是什么，get 都会读取。具体地说，无论下个输入字符是空白字符(空格、制表符等)，还是换行符，get 都会读取它。get 函数获取 char 类型的变量作为参数。调用 get 时，会读取下一个输入字符，并把实参变量(下文称为 *Char_Variable*)设为该字符：

语法

```
inputStream.get(Char_Variable);
```

示例

```
char nextSymbol;
cin.get(nextSymbol);
```

要用 get 从文件读取输入，就用输入文件流代替 cin 流。例如，假定 inStream 是文件输入流，以下语句从输入文件读取一个字符并把它放到 char 变量 nextSymbol 中：

```
inStream.get(nextSymbol);
```

为输入文件流(比如 inStream)使用 get 之前，程序必须先调用 open 函数将流连接到输入文件。

成员函数 put 类似于成员函数 get，只是它用于输出。put 允许程序输出一个字符。它获取一个参数(char 类型的表达式，比如 char 类型的常量或变量)。调用该成员函数后，参数的值会被输出到流。例如，以下语句将向屏幕输出字母'a'：

```
cout.put('a');
```

用以前讨论的方法做不了的事情，用 cout.put 一样做不了。只是考虑到知识的完整性，才简单介绍一下它。

如程序使用了 cin.get 或 cout.put，那么与 cin 和 cout 的其他用法一样，需在程序中包含以下预编译指令：

```
#include <iostream>
```

类似地，如程序要为输入文件流使用 get，或者要为输出文件流使用 put，那么和其他任何文件 I/O 一样，需在程序中包含以下预编译指令：

```
#include <fstream>
```

使用以上任何一个 include 指令，还需同时包含以下 using 语句：

```
using namespace std;
```

成员函数 put

每个输出流都有名为 put 的成员函数，它获取一个 char 类型的参数。调用成员函数 put 后，它的参数(后文称之为 *Char Expression*)的值被输出到输出流。

语法

```
outputStream.put(Char_Expression);
```

示例

```
cout.put(nextSymbol);
cout.put('a');
```

要用 put 向文件输出，就用输出文件流代替 cout 流。例如，假定 outStream 是文件输出流，则以下代码将字符'Z'输出到与 outStream 流连接的文件：

```
outStream.put('Z');
```

为输出文件流(比如 outStream)使用成员函数 put 之前，程序必须先调用 open 函数将流连接到输出文件。

putback 成员函数(选读)

有时需要知道输入流中的下个字符。但在读取了下个字符之后，却发现自己不想处理该字符，所以想把它重新放回输入流。例如，假定希望程序一直读取一个输入流中的字符，直到(但不包括)第一个空格。所以，程序必须读取第一个空格，否则不知道什么时候应该停止读取。但另一方面，既然已读取了该空格，它就不再包含在输入流中。与此同时，程序其他部分可能需要读取和处理这个空格。有许多方案都能解决这个问题，但最简单的方案就是使用成员函数 putback。putback 是每个输入流都有的成员函数。它获取一个 char 参数，并将该参数的值放回输入流。该参数可以是能求值为 char 值的任何表达式。

例如，以下代码从与输入流 fin 连接的文件读取字符，将它们写入与输出流 fout 连接的另一个文件。代码一直读取字符，直到(但不包括)第一个空格：

```
fin.get(next);
while (next != ' ')
{
    fout.put(next);
    fin.get(next);
}
fin.putback(next);
```

注意，执行完这段代码之后，已被读取的空格仍然包含在输入流 fin 中，因为代码在读取了这个空格之后，又把它放回原来的输入流中。还要注意，putback 是将一个字符放回输入流中，而 put 是将一个字符放到输出流中。被成员函数 putback 放回输入流的字符不一定是上次读取的字符，它可以是任意字符。putback 是将字符放回输入“流”而不是放回输入“文件”，原始输入文件的内容不发生任何改变。

编程实例　检查输入

如用户输入不正确，再运行程序没有任何价值。为了使程序对不正确输入“免疫”，输入函数应允许用户重新输入，直到输入正确为止。图 6.7 的 getInt 函数询问用户输入是否正确。如表示输入不正确，就要求输入新值。图 6.7 只是测试 getInt 函数的一个驱动程序。但在需要从键盘获取输入的几乎任何程序中，都可利用该函数或其他类似函数。

图 6.7　检查输入

```
1  // 演示 newLine 和 getInt 函数
2  #include <iostream>
3  using namespace std;
4
5  void newLine();
6  // 丢弃当前输入行上剩余的所有输入
7  // 同时丢弃行末的'\n'
8  // 这个版本只支持键盘输入
9
10 void getInt(int& number);
11 // 后条件：为 number 变量
12 // 赋一个由用户核准的值
13
14
15 int main()
16 {
17     int n;
18
19     getInt(n);
```

```
20     cout << "Final value read in = " << n << endl
21          << "End of demonstration.\n";
22     return 0;
23 }
24
25
26 // 使用 iostream:
27 void newLine()
28 {
29     char symbol;
20     do
31     {
32         cin.get(symbol);
33     } while (symbol != '\n');
34 }
35 // 使用 iostream:
36 void getInt(int& number)
37 {
38     char ans;
39     do
40     {
41         cout << "Enter input number: ";
42         cin >> number;
43         cout << "You entered " << number
44              << " Is that correct? (yes/no): ";
45         cin >> ans;
46         newLine();
47     } while ((ans != 'Y') && (ans != 'y'));
48 }
```

示范对话

```
Enter input number: 57
You entered 57 Is that correct? (yes/no): No
Enter input number: 75
You entered 75 Is that correct? (yes/no): yes
Final value read in = 75
End of demonstration.
```

注意对 newLine()函数的调用。该函数读取当前行上剩余的所有字符，但不对它们进行任何处理。这相当于丢弃了该行剩余的所有字符。因此，如用户输入 No，程序就读取第一个字母，也就是 N，再调用 newLine 函数，该函数丢弃输入行中剩余的所有字符。这表示如果用户在下一行输入 75，如示范对话所示，程序将读取数字 75，但不会读取 No 中的字母 o。如程序不包含 newLine 函数调用，那么读取的下一项就是包含 No 的那一行上的 o，而非下一行的数字 75。

注意，getInt 函数用于终止 do-while 循环的布尔表达式。如发现输入不正确，用户应输入 No(或其他形似单词，比如 no)，这会导致再次循环(以便重新输入)。但是，我们的 do-while 循环不是检查用户输入的单词是否以'N'开头，而是检查用户应答的第一个字母是否不等于'Y'(包括小写的'y')。如果用户只用 Yes 或 No 的某种形式来应答，那么“检查是否等于 No”和“检查是否不等于 Yes”实际是同一码事。但由于用户可能以其他方式应答，所以“检查是否不等于 Yes”显得更安全。为了体会这一点，可以假定用户在输入一个数字时出错。计算机回显该数字，并询问它是否正确。用户应输入 No，但假设用户此时又犯错，输入了 Bo(因为在键盘上，B 刚好在 N 的旁边)。由于'B'不等于'Y'，所以程序会执行 do-while 循环主体，用户有机会重新输入。

但如果正确应答是 Yes，但用户不慎输入首字母不是'Y'或'y'的单词，又会出现什么情况呢？在这种情况下，本来不该再次循环，但实际会再循环一次。这虽然是错误的，但严重程度不如前一段讨论的错误。用户虽然必须多输入一次数字，但不至于冒险让整个程

序带着错误的输入继续运行。检查输入时，宁可冒险多执行一次循环，也不要冒险带着不正确的输入继续运行程序。

陷阱：输入中不期而遇的'\n'

使用成员函数 get 时，必须考虑到可能输入的每个字符，其中包括那些不被视为符号的字符，比如空格和换行符'\n'。使用 get 时，一个常见的问题是忘记丢弃每行末尾的换行符'\n'。如果输入流中还有一个没有被读取的换行符(它通常应该丢弃)，那么在程序打算使用成员函数 get 读取真正的符号时，它会读取换行符'\n'。要清除输入流中所有剩余的换行符'\n'，可使用图 6.7 定义的 newLine 函数。下面来看一个具体的例子。

不同形式的 cin 可以混合使用，这是合法的。例如，以下语句就是合法的：

```
cout << "Enter a number:\n";
int number;
cin >> number;
cout << "Now enter a letter:\n";
char symbol;
cin.get(symbol);
```

但这段代码存在问题，如以下示范对话所示：

```
Enter a number:
21
Now enter a letter:
A
```

在上述对话中，数字的值是你期望的 21。但是，如果你期望 symbol 的值是'A'，就肯定会失望。赋给 symbol 的值是'\n'。读取数字 21 之后，输入流的下个字符是换行符'\n'，所以读取的下个字符就是换行符。记住，成员函数 get 不会跳过换行符和空白字符(事实上，具体取决于程序中的其他代码，你可能根本没有机会输入 A。一旦为变量 symbol 填充换行符'\n'，程序就会继续执行下一个语句。如果下个语句是将输出发送到屏幕，屏幕上就会被输出的内容填充，而你没有机会输入 A)。

可用以下两种方式修改代码，正确地为变量 number 填充 21，为变量 symbol 填充'A'。

```
cout << "Enter a number:\n";
int number;
cin >> number;
cout << "Now enter a letter:\n";
char symbol;
cin >> symbol;
```

另外，也可使用图 6.7 定义的 newLine 函数，如下所示：

```
cout << "Enter a number:\n";
int number;
cin >> number;
newLine();
cout << "Now enter a letter:\n";
char symbol;
cin.get(symbol);
```

上述代码允许混合使用 cin 的两种形式，程序能正常运行，但使用时要多加小心。■

万用型流参数

如函数要获取输入流实参，而且实参有时是 cin，有时是输入文件流，那么可以使用 istream(没有 f)类型的形参。但即使作为 istream 类型的实参提供给函数，输入文件流也必须声明为 ifstream 类型(有 f)。

类似，如函数要获取输出流实参，而且实参有时是 cout，有时是输出文件流，那么可以使用 ostream(没有 f)类型的形参。但即使作为 ostream 类型的实参提供给函数，输出文件流也必须声明为 ofstream 类型(有 f)。不能打开或关闭 istream 或 ostream 类型的对象。传给函数前打开这些对象，函数调用结束后关闭。

编程实例　另一个 newLine 函数

为体会如何让流函数变得更通用，请思考一下图 6.7 的 newLine 函数。该函数只允许从键盘输入(换言之，只支持来自预定义流 cin 的输入)。图 6.7 的 newLine 函数是无参的。下面重写它来添加一个 istream 类型的形参。

```
// 使用 iostream:
void newLine(istream& inStream)
{
    char symbol;
    do
    {
        inStream.get(symbol);
    } while (symbol != '\n');
}
```

现在假定程序包含这个新版本的 newLine 函数。如程序从名为 fin 的输入流(已连接到一个输入文件)获取输入，以下调用会丢弃正在从输入文件读取的当前行的剩余内容：

```
newLine(fin);
```

另一方面，如果程序还要从键盘读取输入，以下调用会丢弃当前正在从键盘输入的那一行的剩余内容：

```
newLine(cin);
```

如程序只含有 newLine 的这个改写过的版本(它可获取包括 fin 或 cin 在内的实参)，那么始终都要指定流名称，即使流名称就是 cin。但通过函数重载，可在一个程序中使用 newLine 的两个版本：一个无参(如图 6.7 所示)，一个获取 istream 参数。如程序同时包含函数的这两个版本，以下两个调用就是等价的：

```
newLine(cin);
```

和

```
newLine();
```

其实真的没必要使用 newLine 的两个版本。带一个 istream 参数的 newLine 函数可支持所有情况。但许多程序员有这样的观点：专门为键盘输入提供一个无参版本显得更方便，键盘输入毕竟是最常见的。

函数的默认实参(选读)

视频讲解：*Default Arguments*

除了使用 newLine 函数的两个版本，另一个办法是使用**默认实参**。以下代码第三次重写 newLine 函数：

```
// 使用 iostream:
void newLine(istream& inStream = cin)
{
    char symbol;
    do
    {
        inStream.get(symbol);
    } while (symbol != '\n');
}
```

像下面这样调用函数就会取默认实参 cin：

```
newLine();
```

像下面这样调用就取指定实参 fin：

```
newLine(fin);
```

该机制支持任意实参类型和实参数量。

如部分形参有默认实参，部分没有，有默认实参的形参就必须统一放到参数列表末尾。在参数列表中，如果有的形参有默认实参，有的没有，在进行函数调用时，你要么提供数量最少的实参(换言之，为所有没有默认实参的所有形参提供实参)，要么提供全部实参(与形参数目相同)。提供的实参首先依次应用于无默认实参的所有形参，再依次应用于有默认实参的所有形参。

下面是一个例子：

```
// 演示默认实参的行为，
// 使用 iostream
void defaultArgs(int arg1, int arg2, int arg3 = -3, int arg4 = -4)
{
    cout << arg1 << ' ' << arg2 << ' ' << arg3 << ' ' << arg4 << endl;
}
```

调用该函数时可传递 2 个、3 或者 4 个实参。例如以下调用：

```
defaultArgs(5, 6);
```

它只为没有默认实参的形参提供了实参，其他形参使用默认实参，输出如下：

```
5 6 -3 -4
```

接下分析以下调用：

```
defaultArgs(6, 7, 8);
```

它为没有默认实参的形参提供了实参，为有默认实参的第一个形参提供了新的实参，最后一个形参则使用默认实参，所以输出如下：

```
6 7 8 -4
```

以下调用提供了全部 4 个实参：

```
defaultArgs(5, 6, 7, 8);
```

它将指定的 4 个值分别赋给参数列表中的 4 个参数，输出如下：

```
5 6 7 8
```

√ 自测题

22. 假设 c 是 char 变量，以下两个语句有何区别？

```
cin >> c;
```

和

```
cin.get(c);
```

23. 假设 c 是 char 变量，以下两个语句有何区别？

```
cout << c;
```

和

```
cout.put(c);
```

24. (要求已学过本章选读小节“putback 成员函数”)成员函数 putback 将一个符号“放回”(put back)原输入流。放回输入流的符号是否必须是从流读取的最后一个符号？例如，如果程序从输入流读取了一个'a'，可用 putback 函数放回一个'b'吗？还是只能放回一个'a'？

25. 给定以下代码(假定已嵌入一个完整和正确的程序中，并开始运行)：

```
char c1, c2, c3, c4;
cout << "Enter a line of input:\n";
cin.get(c1);
cin.get(c2);
cin.get(c3);
cin.get(c4);
cout << c1 << c2 << c3 << c4 << "END OF OUTPUT";
```

如果对话像下面这样开始，输出的下一行会是什么？

```
Enter a line of input:
a b c d e f g
```

26. 给定以下代码(假定已嵌入一个完整和正确的程序中，并开始运行)：

```
char next;
int count = 0;
cout << "Enter a line of input:\n";
cin.get(next);
while (next != '\n')
{
    if ((count%2) == 0)        <------ 如果 count 为偶数，就为 true
        cout << next;
    count++;
    cin.get(next);
}
```

如果对话像下面这样开始，输出的下一行会是什么？

```
Enter a line of input:
abcdef gh
```

27. 假设运行自测题 26 所描述的程序，而且对话像下面这样开始(而不是像自测题 26 那样开始)，输出的下一行是什么？

```
Enter a line of input:
0 1 2 3 4 5 6 7 8 9 10 11
```

28. 给定以下代码(假定已嵌入一个完整和正确的程序中，并开始运行)：

```
char next;
int count = 0;
cout << "Enter a line of input:\n";
cin >> next;
while (next != '\n')
{
    if ((count % 2) == 0)
        cout << next;
    count++;
    cin >> next;
}
```

如果对话像下面这样开始，输出的下一行会是什么？

```
Enter a line of input:
0 1 2 3 4 5 6 7 8 9 10 11
```

29. 定义名为 copyChar 的函数，取一个输入流作为参数。调用 copyChar 时，函数从作为实参提供的输入流中读取一个字符，将该字符写到屏幕。调用 copyChar 时，要能将 cin 或者一个输入文件流作为该函数的实参(如实参是输入文件流，该流在函数调用之前已经连接到一个文件，所以 copyChar 不必打开或关闭任何文件)。例如以下两个 copyChar 函数调用，第一个从文件 stuff.dat 将一个字符复制到屏幕，第二个从键盘将一个字符复制到屏幕：

```
ifstream fin;
fin.open("stuff.dat");
copyChar(fin);
copyChar(cin);
```

30. 定义名为 copyLine 的函数，取一个输入流作为参数。调用 copyLine 时，函数从作为实参指定的输入流中读取一行输入，并将那一行写到屏幕。调用 copyLine 时，要能将 cin 或一个输入文件流作为该函数的实参(如实参是输入文件流，该流在函数调用之前已经连接到一个文件，所以 copyLine 不必打开或关闭任何文件)。例如以下两个 copyLine 函数调用，第一个从文件 stuff.dat 将一行复制到屏幕，第二个从键盘将一行复制到屏幕：

```
ifstream fin;
fin.open("stuff.dat");
copyLine(fin);
copyLine(cin);
```

31. 定义名为 sendLine 的函数，取一个输出流作为参数。调用 sendLine 时，函数从键盘读取一行输入，再将这一行发送到作为实参指定的输出流。调用 sendLine 时，cout 或一个输出文件流都能作为该函数的实参(如实参是输出文件流，该流在调用函数之前已经连接到一个文件，所以 sendLine 不必打开或关闭任何文件)。例如以下两个 sendLine 函数调用，第一个将来自键盘的一行输入复制到 morestuf.dat 文件，第二个则将来自键盘的一行输入复制到屏幕：

```
ofstream fout;
fout.open("morestuf.dat");
cout << "Enter 2 lines of input:\n";
sendLine(fout);
sendLine(cout);
```

32. (要求已学过本章选读小节“函数的默认实参”)给定以下函数定义，后面三个函数调用输出什么？

```
void func(double x, double y = 1.1, double z = 2.3)
{
    cout << x << " " << y << " " << z << endl;
}
```

三个函数调用如下。

```
a. func(2.0);
b. func(2.0, 3.0);
```

c. func(2.0, 3.0, 4.0);

33. (要求已学过本章选读小节“函数的默认实参”)写 func 函数的几个重载版本，使其效果与前一题使用默认参数的效果相同。

eof 成员函数

每个输入文件流都有名为 eof 的成员函数，用于判断何时读完文件的全部内容，没有更多的输入。这是判断程序是否读完文件全部内容的第二种方法。

eof 是 **end of file** 的缩写，表示文件尾，一般读作 e-o-f。函数无参。对于名为 fin 的输入流，可像下面这样写 eof 函数调用：

```
fin.eof()
```

这是可用来控制 while 循环、do-while 循环或 if-else 语句的布尔表达式。如程序已越过输入文件的末尾，表达式将得到满足(也就是为 true)；否则不满足(也就是为 false)。

由于通常只想测试是否没有抵达文件尾，所以在调用成员函数 eof 时，通常在它前面使用逻辑求反操作符!。例如以下语句：

```
if (! fin.eof())
   cout << "Not done yet.";
else
   cout << "End of the file.";
```

if 语句之后的布尔表达式表示“没有抵达与 fin 连接的文件尾”。因此，假设程序尚未越过连接到 fin 流的那个文件的末尾，上述 if-else 语句会在屏幕上输出以下结果：

```
Not done yet.
```

如程序已越过文件尾，if-else 语句就会输出以下内容：

```
End of the file.
```

作为使用 eof 成员函数的另一个例子，假定 inStream 输入流已通过 open 函数调用连接到一个输入文件。然后，整个文件的内容都可通过以下 while 循环输出到屏幕：

```
inStream.get(next);
while (! inStream.eof())
{
    cout << next;  <-------- 如果愿意，这里也可使用 cout.put(next)
    inStream.get(next);
}
```

while 循环使用成员函数 get 从输入文件将每个字符读入 char 类型的 next 变量，并将它们一一输出到屏幕。程序越过文件尾之后，inStream.eof()的值从 false 变成 true。因此，以下表达式从 true 变成 false，循环终止：

```
(! inStream.eof())
```

注意，除非程序试图读取文件尾之后的一个字符，否则 inStream.eof()不会变成 true。例如，假定一个文件包含以下内容(c 之后没有换行符)：

```
ab
c
```

它实际就是以下 4 个字符：

ab<*换行符*'\n'>c

以上循环读取'a'，将其写到屏幕上，再读取'b'，写到屏幕上，再读取换行符'\n'，写到屏幕上，最后读取'c'，写到屏幕上。至此，循环已读完文件中的所有字符。但此时的 inStream.eof()仍为 false。只有在程序试图继续读取一个字符时，inStream.eof()的值才从 false 变成 true。这就是前面的 while 循环为什么用一个 inStream.get(next)来结束的原因。循环需再读取一个字符才能终止。

文件末尾有一个特殊的“文件尾”标记。成员函数 eof 只有在读取了这个文件尾标记之后，才会从 false 变成 true。这是前面的 while 循环在读取了文件的“最后一个字符”之后还要再读取一个字符的原因。但这个文件尾标记不是普通字符，不能把它与普通字符同样对待。虽然可以读取该标记，但不能写出来。如写出文件尾标记，结果将难以预料。系统自动将文件尾标记放在每个文件的末尾，不用你操心。

6.3 节将使用成员函数 eof 来判断程序何时读完整个输入文件。

现在已掌握了检测文件尾的两种方法。既可使用成员函数 eof，也可使用 6.2 节所描述的方法。大多数情况下，使用任何一种方法都可以，但许多程序员会在不同情况下使用不同方法。如果不知道如何选择，请遵循以下原则：将输入当作文本进行处理，并用成员函数 get 读取输入时，就用成员函数 eof；处理数值数据时，则用另一种方法。

√ 自测题

34. 假定 ins 是文件输入流，它已通过成员函数 open 连接到一个文件。假定程序刚刚读取了文件中的最后一个字符，此时，ins.eof()会求值为 true 还是 false?

35. 为名为 textToScreen 的 void 函数写一个定义。该函数有一个名为 fileStream 的形参，类型是 ifstream。函数的前条件和后条件如下所示：

```
// 前条件：fileStream 流已经通过对成员函数 open 的调用，连接到一个文件
// 后条件：连接到 fileStream 的那个文件中的内容复制到屏幕，每次复制一个字符，
// 所以屏幕输出和文件中文本的内容是一样的
// (该函数不关闭文件)
```

编程实例　编辑文本文件

本节的程序是编辑文本文件的一个非常简单的例子。假定某软件公司利用程序来更新自己的宣传手册。公司已销售过好几个 C 语言编译器，并在近期发布了一系列 C++编译器。该程序可根据现有的 C 宣传材料自动生成 C++宣传材料。程序从含有广告语(阐述 C 编译器的优点)的文件(名为 cad.dat)中读取输入，将类似的广告语(阐述 C++编译器的优点)输出到另一个文件(名为 cplusad.dat)。示范程序如图 6.8 所示。

图 6.8　编辑文本文件

```
1  // 该程序创建名为 cplusad.dat 的文件，该文件和 cad.dat 文件相似，
2  // 只是后者中的所有'C'都被替换为'C++'。假定 cad.dat 中除了代表 C 编程语言
3  // 的 C 之外，没有其他大写字母'C'

4  #include <fstream>
5  #include <iostream>
6  #include <cstdlib>
7  using namespace std;
```

```
 8  void addPlusPlus(ifstream& inStream, ofstream& outStream);
 9  // 前条件：inStream 已通过 open 调用连接到一个输入文件
10  // outStream 也通过 open 连接到一个输出文件
11  // 后条件：与 inStream 连接的文件的内容复制到与 outStream 连接的文件中
12  // 但每个'C'都被替换为"C++"(该函数不关闭文件)
13  int main()
14  {
15      ifstream fin;
16      ofstream fout;
17      cout << "Begin editing files.\n";
18      fin.open("cad.dat");
19      if (fin.fail())
20      {
21          cout << "Input file opening failed.\n";
22          exit(1);
23      }
24      fout.open("cplusad.dat");
25      if (fout.fail())
26      {
27          cout << "Output file opening failed.\n";
28          exit(1);
29      }
30      addPlusPlus(fin, fout);
31      fin.close();
32      fout.close();
33      cout << "End of editing files.\n";
34      return 0;
35  }
36
37  void addPlusPlus(ifstream& inStream, ofstream& outStream)
38  {
39      char next;
40      inStream.get(next);
41      while (! inStream.eof())
42      {
43          if (next == 'C')
44              outStream << "C++";
45          else
46              outStream << next;
47          inStream.get(next);
48      }
50  }
```

cad.dat
(不会被程序更改)

```
C is one of the world's most modern
programming languages. There is no
language as versatile as C, and C
is fun to use.
```

cplusad.dat
(程序运行之后)

```
C++ is one of the world's most modern
programming languages. There is no
language as versatile as C++, and C++
is fun to use.
```

屏幕输出

```
Begin editing files.
End of editing files.
```

程序简单地从 cad.dat 文件读取每一个字符并复制到 cplusad.dat 文件。每个字符原样复制，只是在输入文件中遇到大写字母'C'时，就将字符串"C++"写到输出文件中。程序假设输入文件中的所有大写字母'C'都代表 C 编程语言。因此，为了生成新的宣传材料，只需执行这样的替换就可以了。

注意，从输入文件读取字符并将其写入输出文件时保留了换行符。换行符'\n'和其他字符一样对待。利用成员函数 get 从输入文件中读取换行符后，它会通过插入操作符<<写到输出文件中。必须使用成员函数 get 读取输入。用提取操作符>>读取，程序会忽略所有

空白——程序不会读取输入文件的所有空白字符和换行符，造成它们不会复制到输出文件。

还要注意，成员函数 eof 用于检测输入文件尾和终止 while 循环。

预定义字符函数

进行文本处理时往往需要进行大小写转换。预定义函数 toupper 可将小写字母转换为大写字母。例如，toupper('a')返回'A'。如传给 toupper 的是除了小写字母之外的其他任何东西，它会原封不动地返回这个实参。所以，toupper('A')会返回'A'。函数 tolower 与之相似，区别是将大写字母转换为小写字母。

函数 toupper 和 tolower 都在头文件为 cctype 的库中。所以，只要使用了这些函数或者库中的其他函数，就必须包含以下 include 预编译指令:

```
#include <cctype>
```

图 6.9 对 cctype 库的常用函数进行了总结。

图 6.9　cctype 库的部分预定义函数

函数	说明	示例
toupper(*Char_Exp*)	返回 *Char_Exp* 的大写形式	char c = toupper('a'); cout << c; 输出：A
tolower(*Char_Exp*)	返回 *Char_Exp* 的小写形式	char c = tolower('A'); cout << c; 输出：a
isupper(*Char_Exp*)	如果 *Char_Exp* 是大写字母，就返回 true；否则返回 false	if (isupper(c)) cout << c << " is uppercase."; else cout << c << " is not uppercase.";
islower(*Char_Exp*)	如果 *Char_Exp* 是小写字母，就返回 true；否则返回 false	char c = 'a'; if (islower(c)) cout << c << " is lowercase."; 输出：a is lowercase.
isalpha(*Char_Exp*)	如果 *Char_Exp* 是字母表中的字母，就返回 true；否则返回 false	char c = '$'; if (isalpha(c)) cout << c << " is a letter."; else cout << c << " is not a letter."; 输出：$ is not a letter.
isdigit(*Char_Exp*)	如果 *Char_Exp* 是'0'到'9'的数字，就返回 true；否则返回 false	if (isdigit('3')) cout << "It's a digit."; else cout << "It's not a digit."; 输出：It's a digit.

isspace(*Char_Exp*)	如果 *Char_Exp* 是空白字符(比如空格或换行符)，就返回 true；否则返回 false	// 跳过一个单词，将 c 设为该
		// 单词之后的第一个空白字符；
		do
		{
		   cin.get(c);
		} while (! isspace(c));

isspace 函数在实参是空白字符的前提下返回 true。空白字符是在屏幕显示为空白的任何字符，包括空格、制表符和换行符等。如 isspace 的实参不是空白字符就返回 false。因此，isspace(' ')返回 true，而 isspace('a')返回 false。

例如，以下代码读取以句点结尾的句子，并回显该字符串，其中所有空白字符都被替换为符号'-'：

```
char next;
do
{
    cin.get(next);
    if (isspace(next))     // 如 next 中的字符是空白字符，就返回 true
        cout << '-';
    else
        cout << next;
} while (next != '.');
```

例如，假定为以上代码提供的输入如下：

```
Ahhdo be do.
```

则输出如下：

```
Ahhdo-be-do.
```

陷阱：toupper 和 tolower 返回值

许多方面，C++都将字符视为整数，类似于 int 类型的数字。每个字符都被分配了一个数字。用 char 类型的变量存储字符时，计算机内存实际存储的就是该数字。C++允许将 char 类型的值当作数字使用。例如，可将它放到 int 类型的变量中。还可将 int 类型的数字存储到 char 类型的变量中(前提是数字不能太大)。因此，char 类型既可存储字符，也可存储小的整数。

平时不必关心这个细节，只需将 char 类型的值视为字符，不用关心作为数字使用的情况。但使用 cctype 中的函数时，这个细节就很重要。toupper 和 tolower 函数实际返回 int 类型的值，而不是 char 类型的值。也就是说，它们返回的是与实际字符对应的数字，而不是字符本身。因此，以下语句不会输出字母'A'，而是输出为字母'A'分配的数字：

```
cout << toupper('a');
```

为了让计算机将 toupper 或 tolower 返回的值当作 char 值(而不是 int)进行处理，要事先指定自己需要 char 类型的值。一个办法是将返回值放入 char 类型的变量。以下语句输出字符'A'，通常这正是我们需要的：

```
char c = toupper('a');     // 在变量 c 中放入 'A'
cout << c;
```

要让计算机将 toupper 或 tolower 的返回值当作 char 类型的值进行处理，另一种方式是使用强制类型转换(详情参见 4.2 节“强制类型转换”)，如下所示：

```
cout << static_cast<char>(toupper('a'));     // 输出字符 'A'
```

■

√ 自测题

36. 给定以下代码(假定嵌入一个完整而正确的程序并运行)：

```
cout << "Enter a line of input:\n";
char next;
do
{
   cin.get(next);
   cout << next;
} while ( (! isdigit(next)) && (next != '\n') );
cout << "<END OF OUTPUT";
```

如果对话像下面这样开头，下一行输出是什么？

```
Enter a line of input:
I'll see you at 10:30 AM.
```

37. 写 C++代码读取并回显一行文本，同时删除其中的所有大写字母。

小　　结

- ifstream 类型的流可通过调用成员函数 open 连接到文件。之后就可从文件读取输入。
- ofstream 类型的流可通过调用成员函数 open 连接到文件。之后就可将输出发送到文件。
- 应使用成员函数 fail 检查 open 函数调用是否成功。
- 对象是关联了函数的变量。这些函数称为成员函数。类是其变量为对象的一种类型。例如，一个流是一个对象，而 ifstream 和 ofstream 是类。
- 调用对象的成员函数时，要使用以下语法：

  ```
  Calling_Object.Member_Function_Name(Argument_List);
  ```

  假定 cout 流是调用对象(*Calling_Object*)，precision 是成员函数(*Member_Function_Name*)，对该成员函数的调用可以这样写：

  ```
  cout.precision(2);
  ```

- 流的一些成员函数(比如 width，setf 和 precision)可用于格式化输出。这些输出函数同时适用于连接到屏幕的 cout 流和连接到文件的输出流。
- 每个输入流都有名为 get 的成员函数，用于读取一个字符输入。get 不会忽略空白字符。每个输出流都有名为 put 的成员函数，用于将一个字符写到输出流。
- 成员函数 eof 测试是否抵达输入文件末尾。eof 适合文本处理。但处理数值数据时，建议用本章讨论的另一种方法测试文件尾。
- 函数可以有流类型的形参，但参数只能传引用而不能传值。ifstream 类型可用于输入文件流，ofstream 类型可用于输出文件流(下一条小结描述了其他可选类型)。
- 将 istream(没有 f)用作输入流参数的类型，与该参数对应的实参既可以是 cin 流，也可以是 ifstream(有 f)类型的输入文件流。将 ostream(没有 f)用作输出流参数的类型，与该参数对应的实参既可以是 cout 流，也可以是 ofstream(有 f)类型的输出文件流。

自测题答案

1. 以下是 fin 和 fout 流的声明：

```
ifstream fin;
ofstream fout;
```

要在文件开头添加以下 include 预编译指令：

```
#include <fstream>
```

代码中还应该包含以下语句：

```
using namespace std;
```

2.

```
fin.open("stuff1.dat");
if (fin.fail())
{
    cout << "Input file opening failed.\n";
    exit(1);
}

fout.open("stuff2.dat");
if (fout.fail())
{
    cout << "Output file opening failed.\n";
    exit(1);
}
```

3.

```
fin.close();
fout.close();
```

4. 需要将 outStream 流替换成 cout 流。注意，不需要声明 cout，不需要为 cout 调用 open 函数，也不需要关闭 cout。

5.

```
#include <cstdlib>
```

代码中还应包含以下语句：

```
using namespace std;
```

6. exit(1)函数将它的实参返回给操作系统。根据约定，操作系统用 1 标记出错，用 0 标记成功。具体视系统而定。

7. bla.dobedo(7);

8. 文件和程序变量都可存储值，而且都可从中检索(获取)值。程序变量只存在于程序运行期间，而文件在程序运行前就可能存在，而且在程序终止之后可继续存在。简单地说，文件可以是永久性的，变量则不然。文件能存储大量数据，变量则不能提供如此大的存储量。

9. 在此之前已学习了成员函数 open，close 和 fail。以下代码说明了它们的用法：

```
int c;
ifstream in;
ofstream out;
in.open("in.dat");
if (in.fail())
{
    cout << "Input file opening failed.\n";
    exit(1);
}
in >> c;

out.open("out.dat");
if (out.fail())
{
    cout << "Output file opening failed.\n";
    exit(1);
```

```
}
out << c;

out.close();
in.close();
```

10. 本章开头讲过，这种情况下需“从头读取”。换言之，尚未读完的文件必须关闭并重新打开。只有这样，才能将读取位置重新定位到第一行，以便再次读取之前读取过的内容。

11. 两个名称是外部文件名和流名称。外部文件名是供操作系统使用的文件名，它是文件的真实名称，但只在 open 函数调用中使用一次，该函数将文件连接到一个流。流名称代表一个流变量(通常是 ifstream 或者 ofstream 类型)。调用 open 后，程序只能将流名称作为文件名使用。

12.
```
*  123*123*
*  123*123*
```

每个空白都包含两个空格。注意，width 或 setw 调用只作用于一个输出项。

13.
```
* 123*123  *  123*
```

每个空白都包含两个空格。

14.
```
*  123*123*
* +123*+123*
*123  *123  *
```

第二行中，符号*与+之间只有一个空格。其他每个空白都包含两个空格。

15. 到 stuff.dat 文件的输出与自测题 14 的答案一样。

16.
```
*12345*
```

注意：整个整数都会输出，即使它要求的字符位置超过了 setw 指定的域宽。

17. a. ios::fixed。设置这个标志导致浮点数不用科学记数法(e 记数法)显示。设置这个标志，会取消设置 ios::scientific。

   b. ios::scientific。设置这个标志会导致浮点数用科学记数法(e 记数法)显示。设置这个标志，会取消设置 ios::fixed。

   c. ios::showpoint。设置这个标志，将始终显示浮点数中的小数点和尾随的 0。

   d. ios::showpos。设置这个标志，将导致正数前面输出正号。

   e. ios::right。设置这个标志，而且通过调用成员函数 width 指定了域宽，输出的下一项会对齐指定域的右侧(右对齐)。也就是说，输出项之前会根据需要添加用于填充的空格。设置该标志会自动取消设置 ios::left 标志。

   f. ios::left。设置这个标志，而且通过调用成员函数 width 指定了域宽，输出的下一项会对齐指定域的左侧(左对齐)。也就是说，输出项之后会根据需要添加用于填充的空格。设置该标志会自动取消设置 ios::right 标志。

18. 要用 cout 代替 outstream，并删除为 outstream 进行的 open 和 close 函数调用。cout 不需要声明、打开或者关闭。#include <fstream>预编译指令会包含进行屏幕 I/O 所需的全部 iostream 成员函数。不过，添加#include <iostream>预编译指令也没什么害处，还可以使程序更清晰。

19.
```
1
2
3
3
```

20.
```
void toScreen(ifstream& fileStream)
{
    int next;
    while (fileStream >> next)
        cout << next << endl;
}
```

21. 字符串变量中能输入的最大字符数比声明长度小 1，所以答案是 20。

22. 以下语句将读取下一个非空白字符：

```
cin >> c;
```

但以下语句将读取下一个字符，不管这个字符是不是非空白字符：

```
cin.get(c);
```

23. 两个语句等价。都输出变量 c 的值。

24. 被成员函数 putback 放回输入流的字符不一定是最后一个读取的字符。如程序从输入流读取一个字母'a'，可用 putback 函数放回一个'b'，而不是只能放回'a'(虽然程序表现得好象输入文件的文本发生了改变，但其实没有发生任何改变)。

25. 完整对话如下：

```
Enter a line of input:
a b c d e f g
a b END OF OUTPUT
```

26. 完整对话如下：

```
Enter a line of input:
abcdef gh
ace h
```

注意，是隔一个字符输出。还要注意，空格的处理方式和其他任何字符一样。

27. 完整对话如下：

```
Enter a line of input:
0 1 2 3 4 5 6 7 8 9 10 11
01234567891 1
```

务必注意，输入字符串 10 中只有'1'才会输出。这是因为 cin.get 读取的是字符而不是数字，所以它将输入的 10 当作两个字符'1'和'0'来读取。由于代码的目的是隔一个字符回显一个字符，所以'0'不会输出。由于'0'不会输出，所以会输出它的下一个字符(也就是一个空格)，这就造成输出中出现一个空格。类似地，'11'中的两个字符'1'只输出一个。如果你觉得还不清楚，请拿出一张纸，在上面写好输入，用一个小的正方形表示空格字符。然后，隔一个字符就用叉号划掉一个，最终留下的就是前面显示的输出。

28. 这段代码中包含一个无限循环。只要持续为它提供输入，就会一直运行。布尔表达式 (next != '\n') 始终为 true，因为 next 将通过以下语句进行填充：

```
cin >> next;
```

该这个语句会一直忽略换行符'\n'(和其他任何空白字符)。提供了示例输入后，完整的对话如下：

```
Enter a line of input:
0 1 2 3 4 5 6 7 8 9 10 11
0246811
```

注意，自测题 27 的代码使用 cin.get，所以每个字符都会被读取，不管是不是空白字符，然后隔一个字符输出一个。所以，自测题 27 的代码能真正做到隔一个字符输出，即使是空白字符。相反，本题的代码使用 cin 和>>，会忽略所有空白字符，只能识别非空白字符(本例就是字符'0'到'9')。所以，

这段代码的作用是隔一个非空白字符就输出一个。输出中的两个'1'是输入的 10 和 11 的第一个字符。

29.
```
void copyChar(istream& sourceFile)
{
    char next;
    sourceFile.get(next);
    cout << next;
}
```

30.
```
void copyLine(istream& sourceFile)
{
    char next;
    do
    {
        sourceFile.get(next);
        cout << next;
    } while (next != '\n');
}
```

31.
```
void sendLine(ostream& targetStream)
{
    char next;
    do
    {
        cin.get(next);
        targetStream << next;
    } while (next != '\n');
}
```

32.
```
a. 2.0 1.1 2.3
b. 2.0 3.0 2.3
c. 2.0 3.0 4.0
```

33. 一个函数集为:

```
void func(double x)
{
    double y = 1.1;
    double z = 2.3;
    cout << x << " " << y << " " << z << endl;
}
void func(double x, double y)
{
    double z = 2.3;
    cout << x << " " << y << " " << z << endl;
}
void func(double x, double y, double z)
{
    cout << x << " " << y << " " << z << endl;
}
```

34. 它将求值为 false。为了求值为 true，程序必须在读取了最后一个字符后，尝试多读取一个字符。

35.
```
void textToScreen(ifstream& fileStream)
{
    char next;
    fileStream.get(next);
    while (! fileStream.eof())
    {
        cout << next;
        fileStream.get(next);
    }
}
```

如果愿意，可用 cout.put(next);代替 cout << next;。

36. 完整对话如下:

```
Enter a line of input;
I'll see you at 10:30 AM.
I'll see you at 1<END OF OUTPUT
```

37.
```
cout << "Enter a line of input:\n";
char next;
do
{
    cin.get(next);
    if (!isupper(next))
        cout << next;
} while (next != '\n');
```

注意，应该使用!isupper(next)，而不是使用 islower(next)。这是因为假如 next 包含一个非字母的字符(比如空格或逗号)，则 islower(next)为 false。

编 程 练 习

编程练习一般只需写很小的程序，运用本章提到的编程概念。

1. 写程序搜索含有 int 数字的文件，将这些数字中的最大数和最小数输出到屏幕。文件只包含 int 数字，每个数字以空格或换行符分隔。如果是课堂作业，请向老师索要文件名。

2. 写程序从含有 double 数字的文件中读取输入，将这些数字的平均值输出到屏幕。文件只包含 double 数字，每个数字以空格和/或换行符分隔。如果是课堂作业，请向老师索要文件名。

3. a. 计算一个数据文件的“中位数”(median)。大于中位数的数据元素和小于中位数的数据元素相比，两者在数量上是一样的。考虑到本题的宗旨，假定数据已排好序(升序)。如果文件所含数字的数量为奇数，中位数就是文件中间的那个元素；如果为偶数，中位数就是中间那两个元素的平均数。需要打开文件，统计其中包含多少个数字，然后关闭文件，计算文件的中间位置，然后再次打开文件(记住本章对于“从头读取”的讨论)，计数到你需要的数据项，再计算中位数。

   如果是课堂作业，请向老师索要数据文件来测试自己的程序。否则，就自己创建几个文件，要求一个文件含有奇数个数据项，并按升序排列；另一个文件含有偶数个数据项，同样按升序排列。

   b. 已排好序的文件包含 3 个“四分位数”(quartile)：第一个数有 1/4 的数据值都小于或等于它；第一个和第二个数之间，有 1/4 的数据值；第二个和第三个数之间，也有 1/4 的数据值；最后 1/4 的数据值在第三个数之上。例如，对于 1～100 的整数，25，50 和 75 就是四分位数。请针对你为本项目(a)部分使用的数据文件，找出它的 3 个四分位数。

   提示：应该意识到，完成(a)部分后，就相当于完成了(b)部分的三分之一(因为已经有了第二个四分位数)。同时还应意识到，你几乎已经完成了寻找其余两个四分位数的工作。

4. 写程序从含有 double 数字的文件中读取输入。程序向屏幕输出文件中所有数字的平均数和标准差。文件只包含 double 数字，每个数字以空格或换行符分隔。对于 n_1，n_2，n_3……等数字，它们的标准差定义成下面这些数的平均数的平方根：

   $$(n_1-a)^2,\ (n_2-a)^2,\ (n_3-a)^2,\ \cdots\cdots 等$$

   其中，a 是 n_1，n_2，n_3……等的平均数。如果是课堂作业，请向老师索要文件名。

   提示：编写程序，使其先读取整个文件，计算文件中所有数字的平均数，然后关闭文件，并再次打开文件，计算标准差。

5. 写程序来交流编程心得。程序首先向屏幕写一条心得，要求用户输入另一条心得。然后终止程序。下一个运行该程序的人将接收上一次运行程序的人所输入的心得。心得保存在文件中，每次运行这个程序，文件内容都发生改变。可用文本编辑器将一条初始心得输入文件，使第一个运行程序的人能接收到一条心得。允许用户输入任意长度的心得，行数不限。用户被告知按两次回车键即可终止输入心得。然后，程序检查是否读取了两个连续的字符'\n'，从而测试是否抵达输入的末尾。

6. 写程序从文件中读取文本，将编辑过的文本写入另一个文件。除了连续出现的空格被替换为一个空格之外，编辑过的文本与原始文本没有区别。所以，这个程序唯一的用途就是删除多余空格。程序应定

义一个函数，它接收输入和输出文件流作为参数。如果是课堂作业，请向老师索要文件名。

7. 写程序合并两个文件中的数字，将所有数字写入第三个文件。每个输入文件都包含一组 int 数字，这些数字已按从小到大的顺序排列好。程序运行之后，输出文件应包含一个更长的 int 数字列表，其中包括两个输入文件中的所有数字，而且仍然按从小到大的顺序排列。程序应定义一个函数，它接收两个输入文件流和一个输出文件流作为参数。如果是课堂作业，请向老师索要文件名。

编 程 项 目

编程项目要求综合运用多方面的知识来解决问题，程序一般比编程练习大，解题方式多样化。

1. 写程序生成个性化的垃圾邮件。程序同时从一个输入文件和键盘读取输入。输入文件包含一封邮件的正文，其中用 3 个字符#N#代表收件人姓名。程序要求用户输入一个姓名，将完整邮件写入另一个文件，#N#将被替换为用户输入的姓名。#N#在邮件中只出现一次。

   提示：程序从输入文件读取输入，除非遇到 3 个字符#N#，否则就将它读取的所有内容直接复制到输出文件。一旦遇到#N#，就将输出发送到屏幕，要求用户通过键盘输入收件人姓名。你应该能总结出其他的细节。程序应定义一个函数，它取一个输入文件流和一个输出文件流作为参数。如果是课堂作业，请向老师索要文件名。

   还可以做一个难度稍大的版本(要用到选读小节“文件名作为输入”的知识)，这个版本允许字符串#N#在文件中多次出现(且次数不限)。这种情况下，姓名要用两个字符串变量来存储。对于这个较难的版本，我们假定姓名中只有名字和姓氏，不含中间名和首字母缩写。

2. 写程序计算一门课的成绩(满分 100) 。这门课的记录包含在一个文件中，该文件将用作程序的输入文件。输入文件要严格遵循以下格式：每一行顺序包含一名学生的姓氏，一个空格，学生的名字，又一个空格，最后是学生 10 次测验的分数(全部包含在一行中)。测验分数全是整数，每个分数以空格分隔。程序从这个文件读取输入，将输出发送到另一个文件。输出文件中的数据与输入文件中的数据几乎一样，唯一区别在于输出文件中各行末尾多了一个 double 类型的数字。这个数字是该学生 10 次测验的平均分。如果是课堂作业，请向老师索要文件名。程序至少要使用一个函数，该函数获取文件流作为其全部或部分参数。

3. 对编程项目 2 写的程序进行以下改进。

   a. 每一行的测验分数都允许包含 10 个或 10 个以下的分数(如果少于 10 个，表明学生有缺考的情况)。平均分仍是所有分数的总和除以 10 的结果。对于缺考的测验，学生所得分数为 0。

   b. 输出文件的开始位置，将包含一行(或多行)文本来解释输出。使用格式化指令美化布局。

   c. 将输出放到输出文件之后，程序将关闭所有文件，然后将“输出”的内容复制到“输入”文件，实际效果相当于更改了输入文件的内容。

   程序至少使用两个函数，获取文件流作为其全部或部分参数。如果是课堂作业，请向老师索要文件名。

4. 写程序计算一个文件中包含的单词的平均长度(即每个单词的平均字符数)。一个单词被定义成前后为空白字符、逗号、句点、行首或者行尾的任何一个字符串。程序要定义一个函数来获取一个输入文件流作为参数。该函数还允许将 cin 流作为输入流，虽然本程序不提供 cin 流作为实参。如果是课堂作业，请向老师索要文件名。

5. 写程序纠正 C++程序中的操作符<<和>>用法错误，这两个操作符与 cin 和 cout 配合使用。你的程序将把每个错误的 cin <<替换为正确的 cin >>，把每个错误的 cout >>替换为正确的 cout <<。一个比较简单的版本是，假定每个 cin 及其后面的<<之间肯定有(而且只有)一个空格。类似地，假定每个 cout 及其后面的>>之间肯定有(而且只有)一个空格。

   稍难的版本是允许 cin 与<<之间以及 cout 与>>之间有任意数量的空格，甚至可能根本没有空格。对于这个稍难的版本，要求将 cin 或 cout 及其之后的操作符之间的多个或零个空格替换为一个空格。要更正的程序包含在一个文件中，修改好的版本输出到另一个文件。程序还应定义一个函数，它获取

一个输入文件流和一个输出文件流作为参数。如果是课堂作业，请向老师索要文件名，并问清楚是要做简单版本，还是做稍难的版本。

提示：即使选做稍难的版本，也会发现更容易、更快的办法是先做简单版本，再将其修改为稍难的版本。

6. 写程序允许用户输入任何只有一行的问题，自动回答该问题。程序不会实际注意到问题的内容，但它会读取这个问题，并丢弃它读取的所有内容，而且始终给出以下答案之一：

```
I'm not sure, but I think you will find the answer in Chapter #N.
That's a good question.
If I were you, I would not worry about such things.
That question has puzzled philosophers for centuries.
I don't know. I'm just a machine.
Think about it and the answer will come to you.
I used to know the answer to that question, but I've forgotten it.
The answer can be found in a secret place in the woods.
```

这些答案保存在文件中(每行一个答案)，程序直接从文件中读取下一个答案，并将其作为当前问题的答案来显示。程序读完整个文件之后，会将其关闭，然后重新打开文件，从头开始读取下一个答案。

程序每次输出第一个答案时，必须用 1～18(包括这两个数)的一个数代替符号#N。为了在 1～18 之间选择一个数，程序将一个变量初始化为 18，并在每次输出一个数后，将变量值递减 1，所以章号会从 18 递减至 1。一旦变量值递减为 0，变量值就重新变回 18。请使用 const 修饰符声明全局命名常量 NUMBER_OF_CHAPTERS，将它的值设为 18。

提示：使用本章定义好的 newLine 函数。

7. 这个项目与编程项目 6 差不多，区别在于使用一个更复杂的方法选择答案。程序在读取一个问题时，会统计问题中的字符数，并将这个数字保存到名为 count 的变量中。然后，它用编号为 count % ANSWERS 的答案回答问题。文件中第一个答案的编号是 0，下一个是 1，然后是 2，依此类推。ANSWERS 声明为常量，等于答案文件中的答案个数：

```
const int ANSWER = 8
```

采取这种方式，可以自由更改答案文件，使其包含更多或更少的答案，同时只需要修改常量声明，即可让程序支持不同数量的答案。假定文件中最先列出的答案始终是(即使答案文件发生改变)：

```
I'm not sure, but I think you will find the answer in Chapter #N.
```

用数字替换#N 时，请使用数字 (count % NUMBER_OF_CHAPTERS + 1)，其中的 count 是前面讨论过的变量，NUMBER_OF_CHAPTERS 是上一个编程项目讨论过的全局命名变量，它等于本书的章数(即 18)。

8. 这个程序对文本文件中的各行进行编号。请写一个程序从文件中读取文本，并将读取的每一行同时输出到屏幕和另一个文件，并为其附加一个行号。要在每行的起始处打印行号。行号占用 3 个字符的一个域，并在这个域中右对齐。在行号之后添加一个冒号，再添加一个空格，最后附上原来的文本。每次读取一个字符，而且要忽略每一行原来在开头位置的任何空格。可假定这些行都非常短，能在屏幕上一行之内完整地显示。否则，如果一行文本太长，就要允许默认的打印机或屏幕输出行为(也就是说，要允许自动换行或者截尾)。

一个稍难的版本是事先统计好行数，然后据此判断行号要占用多大的一个域。这个工作是在处理文件中的各个行之前完成的。这个版本要确保每一行的长度不超过 72 个字符；如果一行太长，就需要在刚好能满足这个要求的最后一个完整单词之后插入一个换行符。

9. 写程序计算一个文件的以下各项统计信息：文件中的字符总数、非空白字符的总数和文件中的字母数。将统计结果输出到屏幕和另一个文件。

10. 本书配套资源提供了一个名为 babynames2012.txt 的文本文件。文件包含 2012 年美国最常用的 1000 个男孩和女孩名字。该文件的内容由社会安全局收集整理。

这个以空格分隔的文件总共包含 1000 条记录，每条记录都先列出排名(rank)，然后是对应的男孩名字和女孩名字。最常用的名字最先列出，最不常用的名字最后列出。例如，文件最开头是：

```
1 Jacob Sophia
2 Mason Emma
```

```
3 Ethan Isabella
```

这表明 Jacob 是最常用的男孩名字，而 Sophia 是最常用的女孩名字。Mason 是第二流行的男孩名字，Emma 是第二流行的女孩名字。

写程序允许用户输入一个名字。程序应该读取文件，在女孩和男孩中查找相匹配的名字。如果发现一个匹配项，就输出这个名字的排名。如果没有找到匹配项，程序也应该报告。

例如，假定输入的名字是“Justice”，程序应该输出：

```
Justice is ranked 519 in popularity among boys.
Justice is ranked 518 in popularity among girls.
```

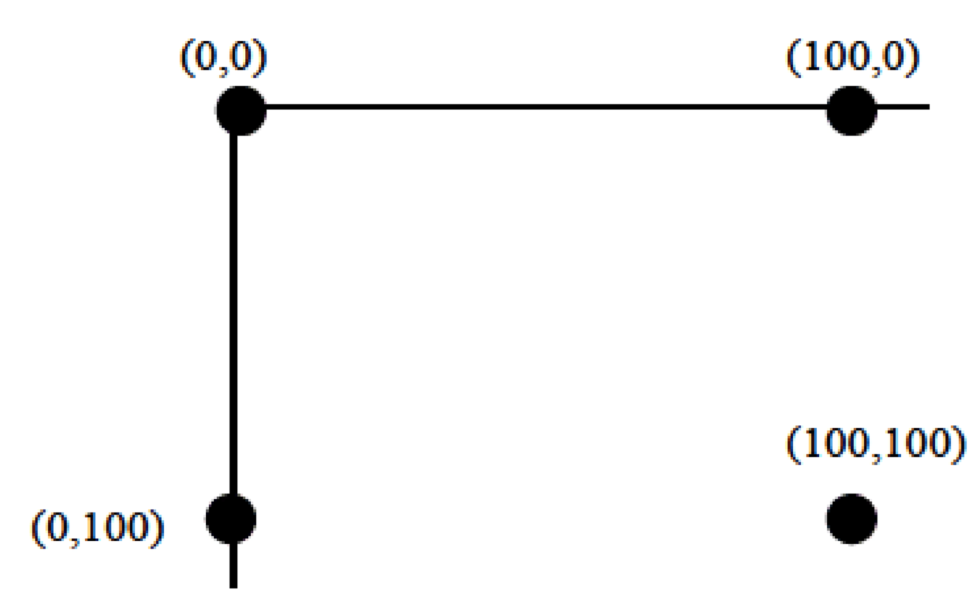
如果输入的名字是“Walter”，程序应该输出：

```
Walter is ranked 376 in popularity among boys.
Walter is not ranked among the top 1000 girl names.
```

11. 视频讲解：*Solution to Programming Project 6.11*

为了完成这道题，你的计算机必须能够查看可扩展矢量图形(Scalable Vector Graphics，SVG)文件。你的浏览器或许已经能够查看这些文件。为了测试浏览器是否能显示 SVG 文件，可以输入下面列出的 `rectline.svg` 文件，并检查是否能在浏览器中打开。如果浏览器不能查看文件，请在网上搜索一下，下载一个免费的 SVG 查看器(推荐使用 Adobe SVG Viewer)。

绘图屏幕使用了一个坐标系统，其中(0,0)位于左上角。x 坐标向右延伸，y 坐标向下延伸。因此，坐标(100,0)在左上角右侧 100 个像素的位置，而坐标(0,100)在左上角下方 100 个像素的位置。如下图所示。



SVG 格式用 XML 定义图像。图像的具体定义数据存储在文本文件中，并可由 SVG 查看器显示。以下是一个示范 SVG 文件，它描绘了两个矩形和一条线。要查看图像，请将代码输入一个文本文件中，并使用.svg 扩展名保存，比如 `rectline.svg`。完成后用 SVG 查看器打开。

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C/DTD SVG 1.1//EN"
"http://www.w3.org/graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="500" height="500"
xmlns="http://www.w3.org/2000/svg">

<rect x="20" y="20" width="50" height="250"
 style="fill:blue;"/>
 <rect x="75" y="100" width="150" height="50"
 style="fill:rgb(0,255,0);"/>
 <line x1="0" y1="0" x2="300" y2="300"
 style="stroke:purple;stroke-width:2"/>

 </svg>
```

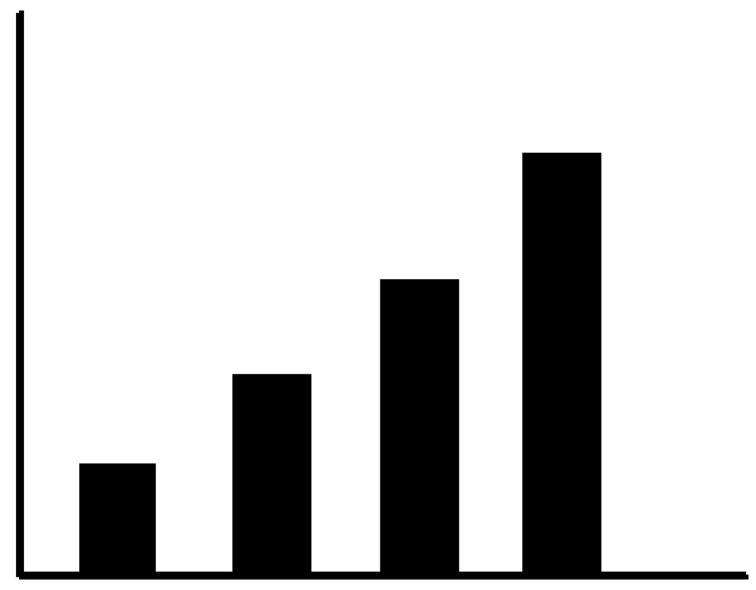
出于本题的宗旨，可以忽略前 5 行和最后一行，把它们视为必须原样输入才能正确创建图像的“样板”。

以<rect x="20"开头的那一行在屏幕上画一个蓝色的矩形，其左上角坐标是(20,20)，宽度为 50 像素，高度为 250 像素。

以<rect x="75"开头的那一行在屏幕上画一个绿色的矩形(RGB 颜色值 0,255,0 对应纯绿色；如果换成 green，则是一种暗绿色)，其左上角坐标是(75,100)，宽度为 150 像素，高度为 50 像素。

最后，<line>标记从(0,0)向(300,300)画一条紫色的线，线宽为 2。

以这个例子为基础，写一个程序来输入 4 个非负的整数值，并创建一个 SVG 文件，显示一幅简单的柱形图来描绘整数值。程序应该按比例增大这些值，确保其中最大的值总是 400 像素高的一个柱形。例如，假定输入的值是 20，40，60 和 120，那么生成的 SVG 文件会显示以下图形。



12. 参考编程项目 11 了解有关 SVG 格式的信息。下面是另一个例子，展示了如何画圆、椭圆和折线：

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C/DTD SVG 1.1//EN"
"http://www.w3.org/graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="500" height="500"
xmlns="http://www.w3.org/2000/svg">

<circle cx="100" cy="50" r="30"
 stroke="green" stroke-width="3" fill="gold"/>

<ellipse cx="100" cy="200" rx="50" ry="100"
style="fill:purple;stroke:black;stroke-width:2"/>

<polyline points="10,10 40,40 20,100 120,140"
style="fill-opacity:0;stroke:red;stroke-width:2"/>

</svg>
```

<circle>标记画一个圆心在(100,50)的圆，半径为 30，笔宽为 3。圆用金色填充，边线为绿色。<ellipse>标记画一个圆心在(100,200)的椭圆，其 x 半径为 30，y 半径为 100。椭圆用紫色填充，边线为黑色。

<polyline>标记画一条红色折线，它从(10,10)到(40,40)到(20,100)到(120,140)。填充不透明度(fill-opacity)设为 0，使屏幕上只显示这条折线，而不显示填充颜色。

基于这些例子以及编程项目 11 的例子，写程序来创建一幅 SVG 图像，它能勾画出你的教授的肖像。线条可以比较简单和抽象。如果想画一幅层次更丰富的图，可以研究一下 SVG 图形格式；还可以使用其他一些标记，在自己的 SVG 图像中运用滤镜(filter)、渐变(gradient)和多边形(polygon)。

13. 写程序提示用户输入一个文本文件的名称，输出文件中的单词数。“单词”被定义成前后为空白字符(空格、回车和换行)、标点符号、文件头和文件尾的任何字符串。

14. 一个古老的字谜是：“猜除了 tremendous、stupendous 和 horrendous 之外其他以 dous 结尾的常见单词。”多想一下，就可能会猜到是什么字。不过，还可以用暴力法解谜。就是读一个包含英语单词的文本文件，输出以 dous 结尾的单词。文本文件“words.txt”含有 87 314 个英语单词，其中包括解这个字谜所需的单词。该文件在本书配套资源中提供。写程序读取文本文件中的每个单词，只输出以 dous 结尾的。

第 7 章　数组

在我们得到这些事实之前就加以推测，那是最大的错误。[①]

——阿瑟·柯南道尔，《福尔摩斯探案集·波希米亚丑闻》

概述

数组用于处理同类型数据的集合，比如温度列表或者名字列表。本章讲述定义和使用C++数组的基础知识，并介绍设计数组相关算法与程序时应掌握的基本技术。

预备知识

本章基于第 2 章～第 6 章的知识。

7.1　数 组 入 门

假定要写程序来读取 5 个考试分数，并对这些分数进行处理。例如，程序可能要计算最高分，然后输出每个分数与最高分的差值。除非读入全部 5 个分数，否则无法知道最高分。所以，5 个分数都要存储下来，计算出最高分后，才能让其他分数与之比较。

为保存 5 个分数，需要与 5 个 int 类型的变量等价的东西。虽然可单独使用 5 个 int 变量，但 5 个变量难以跟踪和维护。另外，以后可能扩展这个程序，比如要支持 100 个分数。在程序中使用 100 个变量显然不现实。数组是解决这种问题的理想方案。**数组**的行为就像一个变量列表，只是采用了统一命名机制，并可用一行简单的代码声明。例如，我们需要的 5 个单独的变量可能是 score[0]，score[1]，score[2]，score[3]以及 score[4]。不变的(本例是 score)是数组名称；可变的是方括号([])内的整数。

声明和引用数组

在 C++中，可以像下面这样声明一个数组，该数组由 5 个 int 类型的变量构成：

```
int score[5];
```

上述声明类似于将以下 5 个变量都声明为 int 类型：

```
score[0],score[1],score[2],score[3],score[4]
```

人们用各种方式称呼数组中独立的变量。我们称为**索引变量**，虽然它们有时也称为**下标变量**或**元素**。方括号中的数字称为**索引**或**下标**。C++索引编号从 0 开始，而不是从 1 或者除 0 之外的其他任何数字开始。数组中的索引变量数目称为数组的**声明长度**，有时也简称为数组**长度**。声明数组时，数组长度在数组名称之后的方括号中给出。然后，索引变量会从 0 开始编号(也使用方括号)，编号结束于数组长度减 1 的整数。

在我们的例子中，索引变量是 int 类型，但数组的索引变量可为任意类型。例如，要声明索引变量类型为 double 的数组，只需在数组声明中将类型名称 int 替换成 double

① 原文是“It is a capital mistake to theorize before one has data”。data 本意是数据，这里引申为事实、证据。——译注

就可以了。但一个数组的所有索引变量必然是同一类型。该类型称为数组的**基类型**。所以，前面的数组 score 的基类型是 int。

可在一行中同时声明数组和普通变量。例如，以下语句除了声明数组 score，还声明了 int 变量 next 和 max：

```
int next, score[5], max;
```

凡是能使用 int 类型的普通变量的地方，都能使用像 score[3]这样的索引变量。

所以，数组名加方括号([])有两种使用方式，不要混淆它们。如果在声明中使用，比如：

```
int score[5];
```

方括号中的数字指定数组含有多少个索引变量。在其他地方使用，方括号中的数字就是一个索引变量的具体索引。例如，score[0]～score[4]都是索引变量。

方括号中的索引不一定是整数常量。可在方括号中使用任何表达式，只要表达式的求值结果在“整数 0”到“数组长度减 1”之间。例如，以下语句将 score[3]的值设为 99：

```
int n = 2;
score[n + 1] = 99;
```

虽然看起来不同，但 score[n + 1]和 score[3]在上述代码中确实是同一个索引变量。这是因为 n + 1 的求值结果是 3。

具体是哪一个索引变量(比如 score[i])由索引(比如 i)的值决定。所以，在程序中可以这样表示：“对索引等于 i 的变量做某事”。其中，i 值由程序来计算。例如，图 7.1 的程序能读取考试分数，按照本章开头的说明处理它们。

图 7.1　使用数组的程序

```
1  // 读入 5 个(考试)分数，显示每个分数与最高分的差值
2  #include <iostream>

3  int main()
4  {
5     using namespace std;
6     int i, score[5], max;

7     cout << "Enter 5 scores:\n";
8     cin >> score[0];
9     max = score[0];
10    for (i = 1; i < 5; i++)
11         {
12       cin >> score[i];
13       if (score[i] > max)
14          max = score[i];
15       // max 是 score[0], ..., score[i]等值中最大的
16    }

17    cout << "The highest score is " << max << endl
18         << "The scores and their\n"
19         << "differences from the highest are:\n";
20    for (i = 0; i < 5; i++)
21       cout << score[i] << " off by "
22            << (max - score[i]) << endl;

23    return 0;
24 }
```

示范对话

```
Enter 5 scores:
5 9 2 10 6
The highest score is 10
The scores and their
differences from the highest are:
5 off by 5
9 off by 1
2 off by 8
10 off by 0
6 off by 4
```

编程提示：为数组使用 for 循环

图 7.1 的第二个 for 循环展示了使用 for 循环来遍历一个数组的常见方式：

```
for (i = 0; i < 5; i++)
    cout << score[i] << " off by "
         << (max - score[i]) << endl;
```

for 语句是操纵数组的理想之选。

陷阱：数组索引总是从零开始

数组索引总是从 0 开始，结束于数组长度减 1 的整数。 ■

编程提示：为数组长度使用已定义常量

再来分析图 7.1 的程序。它只适合刚好 5 名学生的班级，这对现实生活中的大多数班级来说都不合适。为增强灵活性，一个办法是为每个数组的长度都使用已定义的常量。例如，图 7.1 的程序可用以下已定义常量来重写：

```
const int NUMBER_OF_STUDENTS = 5;
```

然后，含有数组声明的那一行要修改成：

```
int i, score[NUMBER_OF_STUDENTS], max;
```

当然，使用数组长度 5 的任何地方都应修改，用 NUMBER_OF_STUDENTS 代替 5。这样一来，如果以后想重写这个程序，使其用于其他任何数量的学生，只需更改定义常量 NUMBER_OF_STUDENTS 的那一行。

注意，不能为数组长度使用变量，例如：

```
cout << "Enter number of students:\n"
cin >> number;
int score[number]; // 在大多数编译器上非法!
```

有的编译器(但并非全部)允许用变量指定数组长度。即使目前所用的编译器允许，为了移植性也不应该这么做 。(第 9 章将讨论另一种数组，它的长度可以在程序运行时确定。)

数 组 声 明

语法

TypeName ArrayName[Declared_Size];

示例

```
int bigArray[100];
double a[3];
double b[5];
char grade[10], oneGrade;
```

上述任何形式的数组声明都会定义总共 *Declared_Size* 个索引变量，这些索引变量从 *ArrayName*[0]开始，一直到 *ArrayName*[*Declared_Size* – 1]。每个索引变量都是一个 *TypeName* 类型的变量。

例如，数组 a 由索引变量 a[0]，a[1]和 a[2]构成，所有变量都是 double 类型。数组 b 由索引变量 b[0]，b[1]，b[2]，b[3]和 b[4]构成，所有变量都是 double 类型。可以在同一行中混合数组声明与简单变量(比如前面的 oneGrade)声明。

数组在内存中的表示

视频讲解：*Array Walkthrough*

讨论数组在计算机内存中如何表示之前，先来研究简单变量(比如 int 或 double 类型的变量)在计算机内存中如何表示。计算机内存由一组编号位置构成，这些位置称为字节[①]。一个字节的编号就是它的**地址**。作为内存一部分来实现的简单变量由一定数量的相邻字节构成。确切的字节数变量类型决定。所以，一个简单变量在内存中要用两项信息来描述：内存地址(那个变量所用的第一个字节的地址)以及变量类型，后者指出该变量需要在内存中占用多少个字节。说“变量的地址”时，指的就是这个地址。程序将值存储到变量时，实际发生的事情是：指定的值(编码成 0,1 序列)被放到内存分配给那个变量的字节中。类似地，变量作为函数的(传引用)实参传递时，实际传给调用函数的是那个变量的地址。下面继续讨论数组在内存中如何存储。

数组的索引变量(数组元素)在内存中采用的表示方法和普通变量一样，但情况要稍微复杂一些。数组各索引变量的位置在内存中肯定相邻。例如以下数组：

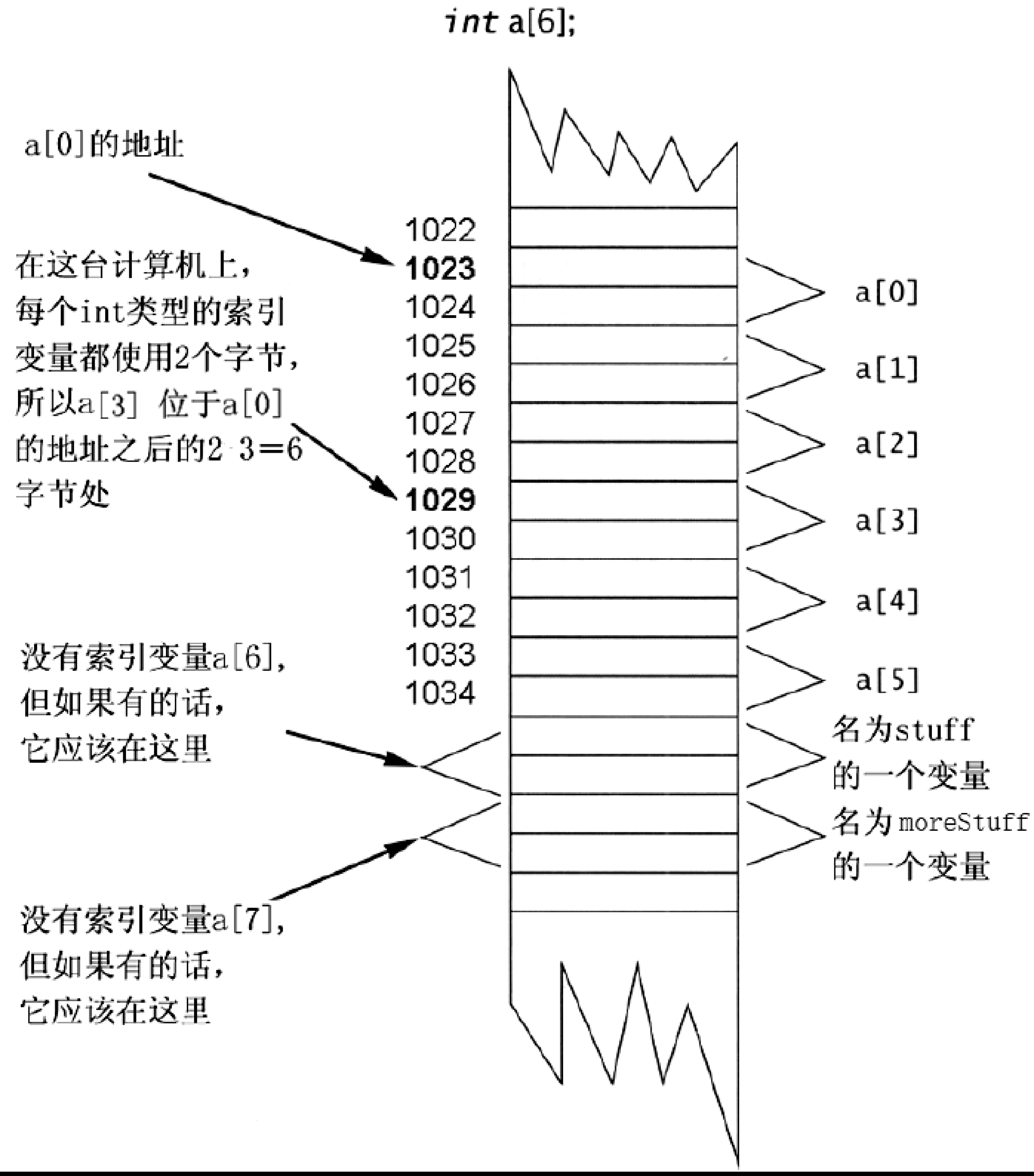
```
int a[6];
```

声明该数组时，计算机会预留容纳 6 个 int 类型的变量所需的那么大的内存。此外，计算机肯定会在内存中一个接一个地存储这些变量。然后，计算机记住索引变量 a[0]的地址，但不会记住其他任何索引变量的地址。程序需要其他索引变量的地址时，计算机会根据 a[0]的地址推算其他索引变量的地址。例如，从 a[0]的地址开始，经过 3 个 int 类型的变量占用的内存空间之后，就来到了 a[3]的地址。所以，为了获得 a[3]的地址，计算机会首先取得 a[0]的地址(这是一个数字)。然后，计算机以 a[0]的地址为基础，加上容纳 3 个 int 类型的变量所需的字节数，就得到了 a[3]的地址。图 7.2 展示了这一过程。

C++数组有许多怪异的特性，但只需理解内存操作的细节，一切难题都会迎刃而解。例如在下一节，将利用这些细节来解释当程序使用一个非法数组索引时发生的事情。

① 1 个字节包含 8 个二进制位，但就目前的讨论来说，字节的准确长度并不重要。

图 7.2　数组在内存中的表示



陷阱：数组索引越界

使用数组时，最常见的错误就是引用不存在的数组索引。例如以下数组声明：

```
int a[6];
```

使用数组 a 时，每个索引表达式都只能求值为 0～5 的整数。例如，假定程序使用了索引变量 a[i]，那么 i 只能求值为 6 个整数之一：0，1，2，3，4 或者 5。i 求值为其他任何结果都会出错。一旦索引表达式的求值结果在数组声明允许的范围之外，就说索引越界或者超出范围，或者简单地说索引**非法**(illegal)。大多数系统上，非法数组索引的结果是程序采取一些错误的操作，这也许是灾难性的错误，而且不会向你发出任何警告。

攻击者利用这种错误来攻陷软件。越界错误可能损害整个系统，所以一定要特别留心防范。2011 年，Common Weakness Enumeration (CWE)/SANS Institute 认定此类错误在最危险程序员错误中名列第三。

例如，假定使用的就是这样一个系统，而且像前面那样声明数组，同时程序中包含以下代码：

```
a[i] = 238;
```

现在，假定 i 不幸地等于 7，那么计算机会正常运行，将 a[7]视为一个合法的索引变量。计算机会计算出 a[7]的地址(似乎真的有 a[7]一样)，并将值 238 放到那个内存位置。但由

于索引变量 a[7]实际是不存在的，获得值 238 的内存位置也许属于另一个变量(假定是一个名为 moreStuff 的变量)。这样，moreStuff 的值就被无故地改变了，如图 7.2 所示。

数组处理循环中，数组索引越界往往发生在第一次迭代或最后一次迭代时。所以必须仔细检查所有数组处理循环，确定开始和结束于合法数组索引。

将数组索引保持在有效范围听着容易做起来难，因为经常不小心或出乎意料地更改索引变量。例如，以下代码将一些数字输入数组：

```
int num;
int a[10];

cout << "How many numbers? (max of 10)" << endl;
cin >> num;
for (int i = 0; i <= num; i++)
{
    cout << "Enter number " << i << endl;
    cin >> a[i];
}
```

发生了两个错误。第一，循环发生“相差 1”错误。索引始于 0，终于 num，造成循环输入 num+1 个数而不是 num 个。为 num 输入的值小于 10 可能注意不到该问题。程序不会崩溃，因为输入的最后一个数的索引不会大于 9。但如果为 num 输入 10，则第 11 个数会存储在元素 a[10]中，偏移数组尾部一个位置。解决方案是这样写循环：

```
for (int i = 0; i < num; i++)
```

第二，缺少输入校验。恶意或懵懂的用户可能为 num 输入 100，造成循环执行 101 次，而执行到第 10 次数组就已越界，之后任何时候都可能造成程序崩溃。解决方案是校验用户输入在有效范围中：

```
cout << "How many numbers? (max of 10)" << endl;
cin >> num;
cout << num << endl;
if (num <= 10)
{
    for (int i = 0; i < num; i++)
    {
        cout << "Enter number " << i << endl;
        cin >> a[i];
    }
}
```

即使这个修订版本也可能出错。为 num 输入的值超过 int 的最大范围会造成溢出。例如，大多数系统的 int 允许存储的最大整数是+2 147 483 647。输入更大的值会溢出，造成在 num 中存储 0 或负值。虽然 num 为零或负时 for 循环不运行，但程序会错误地通过 if 判断。第 8 章会再次探讨这种类型的错误。■

初始化数组

数组可在声明时初始化。初始化数组时，各索引变量的值封闭在花括号内，以逗号分隔。例如：

```
int children[3] = {2, 12, 1};
```

上述声明等价于以下代码：

```
int children[3];
children[0] = 2;
children[1] = 12;
children[2] = 1;
```

如提供的值少于索引变量个数，有多少值就初始化多少个索引变量，剩余索引变量初始化为数组基类型的*零值*。就本例来说，未获得初始值的索引变量会被初始化成整数 0。但在函数(包括 main)定义中，如果只是声明一个数组或其他变量，同时不提供任何初始值，它们将不会初始化。虽然数组索引变量(和其他变量)有时会自动初始化成*零值*，但不应依赖于这种行为。主动初始化一切，才是你的最佳选择。

数组在声明时初始化可省略数组长度。数组自动具有与初始值个数相同的长度。例如，以下声明：

```
int b[] = {5, 12, 11};
```

等价于：

```
int b[3] = {5, 12, 11};
```

编程提示：C++11 基于范围的 for 语句

视频讲解：*Range-Based For Loop*

C++11 支持一种新 for 循环，即基于范围的 for 循环，能简化对数组元素的遍历。语法如下所示：

```
for (datatype varname : array)
{
   // 每个数组元素的值依次赋给 varname
}
```

例如：

```
int arr[] = {2, 4, 6, 8};
for (int x : arr)
   cout << x;
cout << endl;
```

输出结果是 2468。

定义用于遍历数组的变量时，可以使用和定义普通函数参数一样的修饰符。本例定义的 x 变量相当于传值参数。在循环内部更改 x 不会更改数组。但如果用&将 x 定义成传引用，对 x 的修改就会反映到数组。还可用 const 修饰符指定变量不能修改。下例递增数组的每个元素并输出。输出循环则用 auto 数据类型自动判断数组元素类型。

```
int arr[] = {2, 4, 6, 8};
for (int& x : arr)
    x++;
for (auto x : arr)
    cout << x;
cout << endl;
```

输出结果是 3579。基于范围的 for 循环遍历向量(第 8 章介绍)和容器(第 18 章介绍)时特别方便。

√ 自测题

1. 说明 int a[5]和 a[4]的区别，解释[5]和[4]在每种情况下的含义。
2. 在以下数组声明中：

   ```
   double score[5];
   ```

   请说明以下几点：
   a. 数组名称；
   b. 基类型；
   c. 数组的声明长度；
   d. 数组的索引值范围；
   e. 数组的某个索引变量(或元素)。
3. 指出以下数组声明的错误(如果有的话)：

   ```
   a. int x[4] = { 8, 7, 6, 4, 3 };
   b. int x[] = { 8, 7, 6, 4 };
   c. const int SIZE = 4;
   d. int x[SIZE];
   ```

4. 以下代码输出什么？

   ```
   char symbol[3] = {'a', 'b', 'c'};
   for (int index = 0; index < 3; index++)
        cout << symbol[index];
   ```

5. 以下代码输出什么？

   ```
   double a[3] = {1.1, 2.2, 3.3};
   cout << a[0]  << " " << a[1] << " " << a[2] << endl;
   a[1] = a[2];
   cout << a[0]  << " " << a[1] << " " << a[2] << endl;
   ```

6. 以下代码输出什么？

   ```
   int i, temp[10];
   for(i = 0; i< 10; i++)
      temp[i] = 2*i;
   for(i = 0; i < 10; i++)
      cout << temp[i] << " ";
   cout << endl;

   for (i = 0;i < 10; i = i + 2)
      cout << temp[i] << " ";
   ```

7. 以下代码有什么错误？

   ```
   int sampleArray[10];
   for (int index = 1; index <= 10; index++)
      sampleArray[index] = 3 * index;
   ```

8. 假定要对数组 a 的元素进行排序，使下式成立：

   ```
   a[0] ≤ a[1] ≤ a[2] ≤ . . .
   ```

   但为了保险，希望程序测试数组，发现顺序不对的元素就发出警告。以下代码用于输出警告，但它包含一个 bug，请找出该 bug：

```
double a[10];
   <用于填充数组的代码放到这里>
for (int index = 0; index < 10; index++)
   if (a[index] > a[index + 1])
      cout << "Array elements " << index << " and "
           << (index + 1) << " are out of order.";
```

9. 写 C++代码填充数组。该数组含有从键盘读入的 20 个 int 值。不需要写完整程序，只需写出负责这一部分功能的代码，但必须给出数组和所有变量的声明。

10. 假定程序含有以下数组声明：

```
int yourArray[7];
```

另外，假定在你的 C++实现中，int 类型的变量占用 2 个字节的内存。运行程序时，该数组会占用多大内存？假定在运行程序时，系统将内存地址 1000 分配给索引变量 yourArray[0]。那么，索引变量 yourArray[3]的地址是什么？

7.2　函数中的数组

数组索引变量和整个数组都可作为函数参数使用。下面先讨论作为参数使用的数组索引变量。

索引变量作为函数参数

索引变量作为函数参数使用时，用法和其他变量一样。例如，假定程序包含以下声明：

```
int i, n, a[10];
```

如果函数 myFunction 获取 int 参数，那么以下语句合法：

```
myFunction(n);
```

和 n 一样，数组 a 的索引变量也是 int 变量。所以以下语句也合法：

```
myFunction(a[3]);
```

索引变量作为实参使用时要注意一个问题。例如以下函数调用：

```
myFunction(a[i]);
```

如果 i 的值是 3，实参就是 a[3]。另一方面，如果 i 值是 0，上述调用就等价于：

```
myFunction(a[0]);
```

总之，会先对索引表达式进行求值，确定具体哪个索引变量作为实参。

图 7.3 是将索引变量作为函数参数使用的一个简单例子。假定一家小公司有 3 个员工，程序为每个员工都增加 5 天额外的假期。程序十分简单，但确实演示了索引变量如何作为函数的实参使用。注意 adjustDays 函数，它接受名为 oldDays 的 int 参数。程序的 main 部分向函数传递实参 vacation[number]，number 值每次循环都发生变化。注意形参 oldDays 没有任何特别之处，就是一个普通的 int 参数，而 int 是 vacation 数组的基类型。图 7.3 的索引变量是传值参数，但同样的道理也适合传引用参数。索引变量既可用作传值参数，也可用作传引用参数。

图 7.3 索引变量作为实参传递

```
1  // 演示索引变量作为实参传递
2  // 在每个员工允许的放假天数上再加 5 天
3  #include <iostream>
4  const int NUMBER_OF_EMPLOYEES = 3;
5  int adjustDays(int oldDays);
6  // 返回 oldDays 加 5
7  int main()
8  {
9     using namespace std;
10    int vacation[NUMBER_OF_EMPLOYEES], number;
11    cout << "Enter allowed vacation days for employees 1"
12         << " through " << NUMBER_OF_EMPLOYEES << ":\n";
13    for (number = 1; number <= NUMBER_OF_EMPLOYEES; number++)
14       cin >> vacation[number-1];
15    for (number = 0; number < NUMBER_OF_EMPLOYEES; number++)
16       vacation[number] = adjustDays(vacation[number]);
17    cout << "The revised number of vacation days are:\n";
18    for (number = 1; number <= NUMBER_OF_EMPLOYEES; number++)
19       cout << "Employee number " << number
20            << " vacation days = " << vacation[number-1] << endl;
21    return 0;
22 }
23 int adjustDays(int oldDays)
24 {
25    return (oldDays + 5);
26 }
```

示范对话

```
Enter allowed vacation days for employees 1 through 3:
10 20 5
The revised number of vacation days are:
Employee number 1 vacation days = 15
Employee number 2 vacation days = 25
Employee number 3 vacation days = 10
```

√ 自测题

11. 假设有以下函数定义:

```
void tripler(int& n)
{
   n = 3 * n;
}
```

以下哪些函数调用是可以接受的?

```
int a[3] = {4, 5, 6}, number = 2;
tripler(number);
tripler(a[2]);
tripler(a[3]);
tripler(a[number]);
tripler(a);
```

12. 以下代码有什么错误? tripler 的定义在自测题 11 中给出。

```
int b[5] = {1, 2, 3, 4, 5};
for (int i = 1; i <= 5; i++)
```

```
tripler(b[i]);
```

整个数组作为函数参数

视频讲解：*Passing Arrays to Functions*

函数可获取整个数组作为参数。这样一旦调用函数，用于替换形参的就是一个完整数组。但这既不是传值参数，也不是传引用参数，而是所谓的**数组参数**(array parameter)。先分析一个例子。

图 7.4 定义的函数有一个数组形参 a。调用函数时，该参数被替换成一个完整数组。还有一个普通的传值参数(size)，是与数组长度相等的整数值。函数使用从键盘输入的值填充传入的数组(填充其所有索引变量)，然后在屏幕上输出一条消息，指出使用的最后一个数组索引。

图 7.4　获取数组参数的函数

函数声明

```
1  void fillUp(int a[], int size);
2  // 前条件：size 是数组 a 的声明长度
3  // 用户总共需要输入 size 个整数
4  // 后条件：利用从键盘输入的 size 个整数来填充数组 a
```

函数定义

```
1  // 使用 iostream:
2  void fillUp(int a[], int size)
3  {
4      using namespace std;
5      cout << "Enter " << size << " numbers:\n";
6      for (int i = 0; i < size; i++)
7          cin >> a[i];
8      size--;
9      cout << "The last array index used is " << size << endl;
10 }
```

形参 ***int*** a[]是数组参数。看到空的方括号(其中没有索引表达式)，C++就知道这是数组参数。数组参数虽然不是真正的传引用参数，但实际行为和传引用参数相似。下面仔细分析这个例子，体会数组参数在其中的具体运用。注意，**数组实参**(array argument)是替换 a[]这样的数组形参的实际参数值。

调用 fillUp 时必须提供两个实参：第一个是整数数组；第二个是数组的声明长度。例如，下面就是一个可接受的函数调用：

```
int score[5], numberOfScores = 5;
fillUp(score, numberOfScores);
```

上述 fillUp 函数调用会使用从键盘输入的 5 个整数值来填充数组 score。注意在函数声明以及函数定义头中使用的形参 a[]，它必须使用一对空的方括号，不包含索引表达式。(可在数组形参的方括号内插入一个数字，但编译器会忽略那个数字，所以本书不会采用这种做法)。另一方面，实际调用函数时，提供的实参(本例是 score)一定不要添加任何方括号或者任何索引表达式。

在这个函数调用中，数组实参 score 会发生什么事情？简单地说，首先会用实参 score 替换函数主体中的数组形参 a，再执行函数主体。所以，以下函数调用：

```
fillUp(score, numberOfScores);
```

等价于以下代码：

```
{
    using namespace std;
    size = 5;  ◄──────────── 5 是 numberOfScores 的值
    cout << "Enter " << size << " numbers:\n";
    for (int i = 0; i < size; i++)
        cin >> score[i];
    size--;
    cout << "The last array index used is " << size << endl;
}
```

形参 a 有别于我们以前见过的所有形参。形参 a 只是实参 score 的占位符。如果调用 fillUp 函数，并将 score 用作数组实参，计算机的行为就好像是 a 被替换成相应的实参 score。记住：数组作为函数的参数使用时，作用于数组形参的任何操作都会作用于数组实参，所以数组实参的索引变量值可由函数更改。在函数主体中更改形参 (比如使用 cin 语句来更改)，数组实参也会相应更改。

到目前为止，数组形参似乎就是传引用参数。这虽然接近于事实，但两者仍然有少许区别。为理解区别，先回顾一下数组的细节。

以前讲过，数组是作为连续的内存块来存储的。例如以下数组 score 声明：

```
int score[5];
```

声明该数组时，计算机会保留容纳 5 个 int 类型的变量所需的内存。每个变量在计算机内存中依次存储。计算机不会记住全部 5 个索引变量的地址；相反，只记住索引变量 score[0]的地址。例如，当程序需要 score[3]时，计算机根据 score[0]的地址计算出 score[3]的地址。计算机知道 score[3]肯定在 score[0]之后的第 3 个 int 变量处。所以，为了获得 score[3]的地址，计算机会获得 score[0]的地址，在它上面加一个代表“存储 3 个 int 变量所需内存量”的数字，结果就是 score[3]的地址。

从这个角度出发，我们说数组由三部分组成：第一个索引变量的地址(内存位置)、数组基类型(决定了每个索引变量占用的内存量)以及数组长度(索引变量个数)。数组作为数组实参传给函数时，上述三部分只有第一部分才传给函数。因此，当数组实参替换对应的形参时，实际用于替换的是数组第一个索引变量的地址。数组实参的基类型必须与形参基类型一致，所以函数还知道了数组的基类型。但是，数组实参不会将数组长度告诉函数。函数主体中的代码执行时，计算机知道数组从内存的什么位置开始，也知道每个索引变量占用的内存量。但除非你有意指定，否则它不知道数组有多少个索引变量。这正是为什么必须用另一个 int 实参将数组长度告诉函数的原因，也正是为什么说数组形参有别于传引用参数的原因。可将数组形参视为传引用参数的一种弱型(weak form)，因为除了数组长度，它将关于数组的一切都告诉函数。[①]

数组形参看起来可能有点儿怪，但有一个非常显著的优点。为了理解这个优点，最好的办法就是再次分析图 7.4 给出的 fillUp 函数。同一个函数可用来填充任意长度的数组，只要数组的基类型是 int。例如，假定有以下数组声明：

```
int score[5], time[10];
```

以下第一个 fillUp 调用会在数组 score 中填充 5 个值，第二个会在数组 time 中填充 10

① 如果你懂得指针，这看起来就像是在说指针。传递数组实参时，传递的确实是指向它的第一个索引变量(索引为 0 的索引变量)的指针。这方面的详情将在第 9 章讨论。如果还不懂指针，可暂时忽略该注解。

个值：

```
fillUp(score, 5);
fillUp(time,10);
```

由于 size 是单独的参数，所以可向同一个函数传递不同长度的数组实参。

数组形参和实参

可将整个数组传给函数，但该数组参数既非传值参数，也非传引用参数，而是一种新的参数类型，称为**数组参数**。将数组传给函数时，实际传递的是该数组实参的第一个索引变量(索引为 0 的那个)的内存地址。数组实参本身不能告诉函数该数组的长度。所以，函数在接受数组参数时，通常还必须接受代表数组长度的 int 参数，参见稍后的示例。

数组参数与传引用参数的共同点在于：函数主体对数组形参的更改实际会作用于数组实参。所以，函数能更改数组实参的值(也就是更改其索引变量的值)。

以下语法展示了如何在函数声明中使用数组参数：

语法

```
Type_Returned functionName(..., Base_Type  ArrayName[],…);
```

示例

```
void sumArray(double& sum, double a[], int size);
```

const 参数修饰符

向函数传递数组实参，函数能更改存储在数组中的值。有时要求这样。但某些函数定义可能要求不能更改数组，而编码时可能不慎更改了数组中存储的一个或多个值。作为预防措施，可告诉编译器你不希望更改数组实参。这样编译器就会进行检查，确保代码不会因为不慎而更改数组中的任何值。要让编译器知道一个数组实参不能由函数更改，可在相应数组形参之前添加 const 修饰符。用 const 修饰的数组形参称为**常量数组参数**(**constant array parameter**)。

例如以下函数输出数组中的值，但不准备更改它们：

```
void showTheWorld(int a[], int sizeOfA)
// 前条件：sizeOfA 是数组 a 的声明长度
// a 的所有索引变量都已赋值
// 后条件：将 a 中的值写到屏幕
{
    cout << "The array contains the following values:\n";
    for (int i = 0; i < sizeOfA; i++)
        cout << a[i] << " ";
    cout << endl;
}
```

这个函数能很好地工作，但是为了保险起见，应该在函数头添加修饰符 const，如下所示：

```
void showTheWorld(const int a[], int sizeOfA)
```

添加修饰符 const 之后，一旦在函数内部错误更改了数组实参中的任何值，编译器就会报错。例如，下面是 showTheWorld 函数的另一个版本，它错误地更改了数组实参的值。幸好，这个版本的函数定义包括了修饰符 const，所以屏幕上显示的错误消息让我们知道数组 a 被更改了：

```
void showTheWorld(const int a[], int sizeOfA)
// 前条件：sizeOfA 是数组 a 的声明长度
// a 的所有索引变量都已赋值
// 后条件：将 a 中的值写到屏幕
{
   cout << "The array contains the following values:\n";
   for (int i = 0; i < sizeOfA; a[i]++)
      cout << a[i] << " ";
   cout << endl;
}
```

错误，但除非用了修饰符 const，否则编译器不会捕捉它

假如在上述函数定义中没有添加 const 修饰符，而且确实像上面那样犯了错，函数会正常编译并运行，不报告任何错误消息。但代码将包含一个无限循环，会不断递增 a[0]，并向屏幕输出新值。

showTheWorld 的这个版本的问题在于，for 循环中递增的是错误的项。它递增的是索引变量 a[i]，而真正需要递增的是索引 i。在这个不正确的版本中，索引 i 从值 0 开始，而且那个值永远不会改变。相反，实际递增的是 a[i](也就是 a[0])。索引变量 a[i]递增时，数组中的一个值会发生更改。由于包括了修饰符 const，所以计算机会报告一条警告消息。根据那个消息，就知道什么地方出错了。

程序中除了有函数定义，通常还有函数声明。函数定义使用了 const 修饰符，函数声明也必须使用，使函数头和函数声明保持一致。

修饰符 const 适合任何种类的参数，但通常只用于“数组形参”和以类的形式传递的“传引用参数”，后者将在第 11 章讨论。

陷阱：const 参数修饰符的使用不一致

const 参数修饰符要么不用，要么都用。一旦为特定类型的数组形参使用了它，就必须为同一个类型的、不由函数更改的其他所有数组形参使用它。原因和函数内部嵌套的其他函数调用有关。来看看下面的 showDifference 函数定义(其中用到了另一个函数，那个函数的声明在第一行给出)：

```
double computeAverage(int a[], int numberUsed);
// 返回数组 a 的前 numberUsed 个元素的平均值，数组 a 不会改动

void showDifference(const int a[], int numberUsed)
{
   double average = computeAverage(a, numberUsed);
   cout << "Average of the " << numberUsed
        << " numbers = " << average << endl
        << "The numbers are:\n";
   for (int index = 0; index < numberUsed; index++)
      cout << a[index] << " differs from average by "
           << (a[index] - average) << endl;
}
```

上述代码在大多数编译器上都会产生一条错误或警告消息。computeAverage 函数不更改参数 a，但编译器在处理 showDifference 的函数定义时，会认为 computeAverage 确实会(或至少有可能)更改参数 a 的值。这是由于在解释 showDifference 的函数定义时，编译器唯一知道的只有 computeAverage 的函数声明。但在那个函数声明中，没有用 const 告诉编译器参数 a 不会更改。所以，一旦在 showDifference 函数中为参数 a 使用了修饰

符 const，就必须同时将这个修饰符用于 computeAverage 函数中的参数 a。computeAverage 的正确函数声明如下：

```
double computeAverage(const int a[], int numberUsed);
```

■

返回数组的函数

函数不能采用和返回 int 或 double 类型的值一样的方式返回数组。可返回指向数组的指针，从而在一定程度上模拟从函数中返回数组。但我们还没有讲到指针。第 9 章讨论数组与指针的交互时，将讲到如何返回指向数组的指针。在那之前，暂时没办法写一个能返回数组的函数。

案例分析：产量图

在这个案例中，将在一个程序的“自顶向下”(top-down)设计过程中运用数组。针对不同的子任务，会将索引变量以及整个数组作为函数的参数使用。

1. 问题定义

Apex 塑料调匙制造公司(Apex Plastic Spoon Manufacturing Company)委托我们写一个程序来显示条形图，展示其下属 4 个工厂在某一周的产量。工厂的每个部门都单独维护自己的生产数据，这些部门包括茶匙部门、汤匙部门、普通鸡尾酒调匙部门、彩色鸡尾酒调匙部门等。此外，每个厂的部门数量不同。例如，只有一个厂在生产彩色鸡尾酒调匙。每个厂的数据单独输入。每个厂的数据都是一个数字列表，指出那个厂的每个部门的产量。输出的是一个条形图，如下所示(Plant 代表“厂”)：

```
Plant #1 **********
Plant #2 *************
Plant #3 *******************
Plant #4 *****
```

每个星号都代表 1000 件产品。

我们决定为工厂的每个部门都单独读取输入。由于任何部门的产量都不可能为负数，所以可将一个负数用作哨兵值，标记结束输入每个工厂的生产数据。

由于输出以 1000 件为单位，所以产量必须除以 1000。这就带来了一个问题，因为计算机显示的星号数必须是一个整数。例如，它不可能为 1600 件产品显示 1.6 个星号。所以，必须把它取整为最接近的一个 1000 的整数倍数，比如 1600 会取整为 2000，最后输出的是 2 个星号。下面是程序输入/输出的准确陈述。

- 输入：4 个工厂编号为 1～4。要求为每个工厂都输入一个数字列表，其中每个数字都代表那个工厂中的某个部门的产量。列表以一个作为哨兵值的负数结束。
- 输出：一个条形图显示每个工厂的总产量。条形图中的每个星号都相当于 1000 件产量。每个工厂的产量都要取整为最接近的一个 1000 的整数倍数。

2. 问题分析

我们准备使用一个名为 production(产量)的数组，它容纳 4 个工厂中的每一个的总产量。C++数组索引总是从 0 开始。但由于工厂编号是 1～4，而非 0～3，所以不可直接将工

厂编号用作数组索引。相反，需要将编号为 n 的那个工厂的总产量放在索引变量 production[n-1]中。换言之，工厂 1 的总产量是 production[0]，工厂 2 的总产量是 production[1]，依此类推。

由于输出以 1000 件为单位，所以程序必须将数组元素的值除以 1000。假定工厂 3 的总产量是 4040 件，那么 production[2]的值最初被设为 4040。然后，该值除以 1000 并取整，得到 production[2]的终值为 4。所以，在对工厂 3 的总产量进行表示的条形图中，需要输出 4 个星号。

程序的任务可分解为以下子任务。

- inputData：读取每个工厂的输入，使索引变量 production[plantNumber-1]的值等于那个工厂的总产量，其中的 plantNumber 是工厂编号(1～4)。
- scale：针对每个 plantNumber，将索引变量 production[plantNumber-1]的值更改为正确的星号数量。
- graph：输出条形图。

对于执行这些子任务的每个函数，整个 production 数组将作为实参传给它们。作为使用数组形参时的一种惯例，必须再用一个形参来表示数组长度。在本例中，这个长度就是工厂的数目。由于工厂数目是固定的，所以要为它使用一个已定义的常量，该常量将用作 production 数组的长度。图 7.5 展示了程序的 main 部分、用于每个子任务的函数声明以及代表工厂数目的已定义常量。注意，由于没有理由更改传给 graph 函数的数组形参，所以添加了 const 参数修饰符，将那个数组形参变成一个常量参数。图 7.5 展示的只是程序的基本结构，如果将它放到一个独立的文件中，应该能通过编译。这样一来，就可先验证其中是否存在语法错误，再具体定义已经声明的函数。

编译好图 7.5 的文件后，就可以着手定义三个子任务所需的函数。针对每个函数，都要设计一个算法，为函数编写代码，并在测试成功之后设计下一个函数。

图 7.5　图表程序基本结构

```
1   // 读入数据并显示一个条形图，展示每个工厂的产量
2   #include <iostream>
3   const int NUMBER_OF_PLANTS = 4;
4.
5   void inputData(int a[], int lastPlantNumber);
6   // 前条件：lastPlantNumber 是数组 a 的声明长度
7   // 后条件：对于 1～lastPlantNumber 的 plantNumber:
8   // a[plantNumber-1]等于编号为 plantNumber 的那个工厂的总产量
9.
10  void scale(int a[], int size);
11  // 前条件：a[0]～a[size-1]分别有一个非负的值
12  // 后条件：a[i]被更改为 a[i]的原始值除以 1000 并取整的结果，0 <= i <= size-1
13
15  void graph(const int asteriskCount[], int lastPlantNumber);
16  // 前条件：asteriskCount[0]～asteriskCount[lastPlantNumber-1]具有非负的值
17  // 后条件：显示一个条形图，表示编号为 N 的工厂的产量等于 asteriskCount[N-1]乘以 1000,
18  1 <= N <=// lastPlantNumber
19
20  int main()
21  {
22      using namespace std;
23      int production[NUMBER_OF_PLANTS];
```

```
24
25     cout << "This program displays a graph showing\n"
26          << "production for each plant in the company.\n";
27
28     inputData(production, NUMBER_OF_PLANTS);
29     scale(production, NUMBER_OF_PLANTS);
30     graph(production, NUMBER_OF_PLANTS);
31
32     return 0;
33 }
```

3. inputData 的算法设计

inputData 函数的函数声明和注释已在图 7.5 中给出。如程序的 main 部分所示(也由图 7.5 给出)，调用 inputData 时，数组形参 a 会被替换成 production 数组。由于最后一个工厂编号与工厂数目相同，所以形参 lastPlantNumber 会被替换成 NUMBER_OF_PLANTS。inputData 的算法是十分直观的：

对于 1～lastPlantNumber 的 plantNumber，采取以下操作：

读入编号为 plantNumber 的那个工厂的所有生产数据。
累加产量。
将 production[plantNumber-1] 设为累加结果。

4. 编写 inputData 函数

inputData 函数的算法转换成以下代码：

```
// 使用 iostream:
void inputData(int a[], int lastPlantNumber)
{
    using namespace std;
    for (int plantNumber = 1;
        plantNumber <= lastPlantNumber; plantNumber++)
    {
        cout << endl
             << "Enter production data for plant number "
             << plantNumber << endl;
        getTotal(a[plantNumber - 1]);
    }
}
```

代码很简单，因为所有工作都由尚未设计的 getTotal 函数完成。但在讨论 getTotal 函数之前，首先注意 inputData 函数的几个要点。注意，编号为 plantNumber 的那个工厂的生产数据存储在索引为 plantNumber-1 的索引变量中；这是由于数组索引总是从 0 开始，而工厂编号从 1 开始。另外，注意为 getTotal 函数的参数使用了一个索引变量。getTotal 函数负责 inputData 的所有实际工作。

getTotal 函数完成一个工厂的输入工作。它读入那个工厂的生产数据，对数字进行累加，并将累加结果存储到与那个工厂对应的索引变量中。但 getTotal 其实并不关心它的参数是不是索引变量。对于 getTotal 这样的函数，索引变量和其他任何 int 类型的变量无异。所以，getTotal 将使用一个标准的 int 类型的传引用参数。也就是说，getTotal 只是一个普通的输入函数，它和我们讨论数组前看到的那些输入函数没有区别。getTotal 函数读入一个以哨兵值结尾的数字列表，当数字读入时进行累加，并将它的参数(一个 int 类型的变量)的值设为最终的累加结果。对于我们来说，getTotal 函数并无什么新意。图

7.6 显示了 getTotal 和 inputData 的函数定义，并在一个简单测试程序中嵌入了这两个函数。

图 7.6　测试 inputData 函数

```
1   // 测试 inputData 函数
2   #include <iostream>
3   const int NUMBER_OF_PLANTS = 4;
4
5   void inputData(int a[], int lastPlantNumber);
6   // 前条件：lastPlantNumber 是数组 a 的声明长度
7   // 后条件：对于 1~lastPlantNumber 的 plantNumber：
8   // a[plantNumber-1]等于编号为 plantNumber 的那个工厂的总产量
9
10  void getTotal(int& sum);
11  // 从键盘读入非负的整数，并将累加结果放到 sum 中
12
13  int main()
14  {
15      using namespace std;
16      int production[NUMBER_OF_PLANTS];
17      char ans;
18
19      do
20      {
21          inputData(production, NUMBER_OF_PLANTS);
22          cout << endl
23               << "Total production for each"
24               << " of plants 1 through 4:\n";
25          for (int number = 1; number <= NUMBER_OF_PLANTS; number++)
26          cout << production[number - 1] << " ";
27
28          cout << endl
29               << "Test Again?(Type y or n and return): ";
30          cin >> ans;
31      }while ( (ans != 'N') && (ans != 'n') );
32
33      cout << endl;
34
35      return 0;
36  }
37
38  // 使用 iostream：
39  void inputData(int a[], int lastPlantNumber)
40  {
41      using namespace std;
42      for (int plantNumber = 1;
43                plantNumber <= lastPlantNumber; plantNumber++)
44      {
45          cout << endl
45               << "Enter production data for plant number "
46               << plantNumber << endl;
47          getTotal(a[plantNumber - 1]);
48      }
49  }
50
51  // 使用 iostream：
52  void getTotal(int& sum)
53  {
54      using namespace std;
55      cout << "Enter number of units produced by each department.\n"
56           << "Append a negative number to the end of the list.\n";
57
58      sum = 0;
59      int next;
60      cin >> next;
61      while (next >= 0)
62      {
63          sum = sum + next;
```

```
64          cin >> next;
65      }
66
67      cout << "Total = " << sum << endl;
68  }
```

示范对话

```
Enter production data for plant number 1.
Enter number of units produced by each department.
Append a negative number to the end of the list.
1 2 3 -1
Total = 6

Enter production data for plant number 2.
Enter number of units produced by each department.
Append a negative number to the end of the list.
0 2 3 -1
Total = 5

Enter production data for plant number 3.
Enter number of units produced by each department.
Append a negative number to the end of the list.
2 -1
Total = 2

Enter production data for plant number 4.
Enter number of units produced by each department.
Append a negative number to the end of the list.
-1
Total = 0

Total production for each of plants 1 through 4:
6 5 2 0
Test Again?(Type y or n and Return): n
```

5. 测试 inputData 函数

每个函数都应该专门用一个程序测试。在该程序中，那个函数必须是唯一没有测试的函数。inputData 函数包括一个 getTotal 函数调用。所以，应在一个专门的驱动程序中测试 getTotal。全面测试了 getTotal 之后，才能在一个程序(比如图 7.6 的那个程序)中用它来测试 inputData 函数。

测试 inputData 函数时，应该用一个工厂可能出现的各种产量数字来进行测试。测试时，应该包括一个没有任何产量数字的工厂(就像图 7.6 中的工厂 4)，应该包括一个只有一个产量数字的工厂(就像图 7.6 中的工厂 3)，还应该包括一个具有多个产量数字的工厂(就像图 7.6 中的工厂 1 和工厂 2)。非零和零产量数字都应该测试，这正是在图 7.6 中为工厂 2 输入的列表中包括一个 0 值的原因。

6. scale 的算法设计

scale 函数对 production 数组中的每个索引变量的值进行修改，获得要打印的星号个数。由于每个星号都代表 1000 件产品，所以每个索引变量的值都必须除以 1000.0。然后，为了获得星号的个数，结果必须取整为一个最接近的整数。该方法必须能处理任何长度的任何数组 a 的值，所以 scale 的函数声明(最早在图 7.5 中给出，下面进行了复制)必须能支持任意数组 a 和任意 size：

```
void scale(int a[], int size);
```

```
// 前条件：a[0]~a[size-1]分别有一个非负的值
// 后条件：a[i]被更改为 a[i]的原始值除以 1000 并取整的结果，
// 0 <= i <= size-1
```

调用 scale 函数时，数组形参 a 会被替换成数组 production，形参 size 则会被替换成 NUMBER_OF_PLANTS，所以函数调用看起来如下所示：

```
scale(production, NUMBER_OF_PLANTS);
```

scale 函数的算法如下：

```
for (int index = 0; index < size; index++)
    将 a[index]的值除以 1000，并取整为最接近的整数；结果是 a[index]的新值。
```

7. 编写 scale 函数

下面是 scale 的算法转换成 C++代码的结果，其中 round 函数尚未定义，它获取 double 类型的参数，返回 int 值，代表最接近其参数的一个整数。换言之，round 函数将它的参数值取整为最接近的整数。

```
void scale(int a[], int size)
{
    for (int index = 0; index < size; index++)
        a[index] = round(a[index]/1000.0);
}
```

注意，是用 1000.0 来除，而不是用 1000。用 1000 将执行整数除法。例如，2600/1000 结果是 2，而 2600/1000.0 结果是 2.6。虽然本来就想得到取整后的结果，但同时还想进行四舍五入。换言之，希望 2600 在除以 1000 并取整之后，结果是 3，而不 2。

现在分析 round 函数的定义，它将其参数取整为最接近的整数。例如，round(2.3)返回 2，而 round(2.6)返回 3。round 函数以及 scale 函数的代码在图 7.7 中给出。其中，round 的代码需进一步解释。

图 7.7　scale 函数

```
1  // 该程序用于演示 scale 函数
2  #include <iostream>
3  #include <cmath>
4
5  void scale(int a[], int size);
6  // 前条件：a[0]~a[size-1]分别有一个非负的值
7  // 后条件：a[i]被更改为 a[i]的原始值除以 1000 并取整的结果，0 <= i <= size-1
8
9  int round(double number);
10 // 前条件：number >= 0
11 // 返回四舍五入后的整数
12
13 int main()
14 {
15     using namespace std;
16     int someArray[4], index;

17     cout << "Enter 4 numbers to scale: ";
18     for (index = 0; index < 4; index++)
19         cin >> someArray[index];

20     scale(someArray, 4);

21     cout << "Values scaled to the number of 1000s are: ";
22     for (index = 0; index < 4; index++)
23         cout << someArray[index] << " ";
24     cout << endl;
```

```
25     return 0;
26 }
27
28 void scale(int a[], int size)
29 {
30     for (int index = 0; index < size; index++)
31         a[index] = round(a[index]/1000.0);
32 }
33
34 // 使用 cmath:
35 int round(double number)
36 {
37     using namespace std;
38     return static_cast<int>(floor(number + 0.5));
39 }
```

示范对话

```
Enter 4 numbers to scale: 2600 999 465 3501
Values scaled to the number of 1000s are: 3 1 0 4
```

round 函数使用了由头文件为 cmath 的库提供的预定义函数 floor。floor 函数返回刚好小于其参数值的整数。例如，floor(2.1)和 floor(2.9)都返回 2。为了理解 round 如何工作，下面来分析一些例子。以 round(2.4)为例，它的返回值如下：

```
floor(2.4 + 0.5)
```

即 floor(2.9)，结果是 2.0。大于或等于 2.0，但小于 2.5 的任何数字，加上 0.5 的结果肯定小于 3.0。所以，将那个数字加上 0.5 之后传给 floor，返回的是 2.0。因此，对于大于或等于 2.0，但小于 2.5 的任何数字，round 的结果肯定是 2。由于 round 的函数声明要求返回值的类型是 int，所以结果值 2.0 要使用 static_cast<int>强制转型为没有小数点的整数值。

再来分析大于或等于 2.5 的数字，例如 2.6。round(2.6)的返回值如下：

```
floor(2.6 + 0.5)
```

也就是 floor(3.1)，结果是 3.0。大于或等于 2.5，但小于或等于 3.0 的任何数字，加上 0.5 的结果肯定大于 3.0。因此，对于大于或等于 2.5，但小于或等于 3.0 的任何数字，round 的结果肯定是 3。

因此，对于 2.0～3.0 的所有参数值，round 肯定能正常工作。而 2.0～3.0 的参数值无任何特别之处，所以同样的道理也适合所有非负数字。总之，round 肯定能正确处理所有非负参数值。

8. 测试 scale 函数

图 7.7 包含了 scale 函数的一个演示程序，但 round 和 scale 函数的测试程序应该比这个简单的程序更全面。特别是，它们应该允许你多次重新测试，而不是只测试一次。这里不准备给出完整的测试程序，但你首先应该在 round 自己的一个驱动程序中测试它(round 要由 scale 使用)，再用另一个专门的驱动程序来测试 scale。用于测试 round 的程序应测试参数值 0、要“入”(round up)的参数值(比如 2.6)以及要“舍”(round down)的参数值(比如 2.3)。用于测试 scale 的程序应该使用类似的一组数组元素值进行测试。

9．graph 函数

图 7.8 给出了用于生成所需条形图的完整程序。这里不准备逐步教你设计 graph 函数，因为它非常简单易懂。

图 7.8　产量图程序

```
1   // 读入数据并显示一个条形图，展示每个工厂的产量
2   #include <iostream>
3   #include <cmath>
4   const int NUMBER_OF_PLANTS = 4;

5   void inputData(int a[], int lastPlantNumber);
6   // 前条件：lastPlantNumber 是数组 a 的声明长度
7   // 后条件：对于 1～lastPlantNumber 的 plantNumber：
8   // a[plantNumber-1]等于编号为 plantNumber 的那个工厂的总产量

9   void scale(int a[], int size);
10  // 前条件：a[0]～a[size-1]分别有一个非负的值
11  // 后条件：a[i]被更改为 a[i]的原始值除以 1000 并取整的结果，0 <= i <= size-1

12  void graph(const int asteriskCount[], int lastPlantNumber);
13  // 前条件：asteriskCount[0]～asteriskCount[lastPlantNumber-1]具有非负的值
14  // 后条件：显示一个条形图，表示编号为 N 的工厂的产量等于 asteriskCount[N-1]乘以 1000，
15  // 1 <= N <= lastPlantNumber

16  void getTotal(int& sum);
17  // 从键盘读入非负的整数，并将累加结果放到 sum 中

18  int round(double number);
19  // 前条件：number >= 0.
20  // 返回取整为最接近的一个整数的数字

21  void printAsterisks(int n);
22  // 在屏幕上打印 n 个星号

23  int main()
24  {
25      using namespace std;
26      int production[NUMBER_OF_PLANTS];

27      cout << "This program displays a graph showing\n"
28           << "production for each plant in the company.\n";
29      inputData(production, NUMBER_OF_PLANTS);
30      scale(production, NUMBER_OF_PLANTS);
31      graph(production, NUMBER_OF_PLANTS);
32      return 0;
33  }

34  // 使用 iostream：
35  void inputData(int a[], int lastPlantNumber)
    <inputData 剩余的定义与图 7.6 相同>

36  // 使用 iostream：
37  void getTotal(int& sum)
    <getTotal 剩余的定义与图 7.6 相同>

38  void scale(int a[], int size)
    <scale 剩余的定义与图 7.7 相同>

39  // 使用 cmath：
40  int round(double number)
    <round 剩余的定义与图 7.7 相同>
```

```
41  // 使用 iostream:
42  void graph(const int asteriskCount[], int lastPlantNumber)
43  {
44      using namespace std;
45      cout << "\nUnits produced in thousands of units:\n";
46      for (int plantNumber = 1;
47                plantNumber <= lastPlantNumber; plantNumber++)
48      {
49          cout << "Plant #" << plantNumber << " ";
50          printAsterisks(asteriskCount[plantNumber - 1]);
51          cout << endl;
52      }
53  }
54  // 使用 iostream:
55  void printAsterisks(int n)
56  {
57      using namespace std;
58      for (int count = 1; count <= n; count++)
59          cout << "*";
60  }
```

示范对话

```
This program displays a graph showing
production for each plant in the company.
Enter production data for plant number 1.
Enter number of units produced by each department.
Append a negative number to the end of the list.
2000 3000 1000 -1
Total = 6000

Enter production data for plant number 2.
Enter number of units produced by each department.
Append a negative number to the end of the list.
2050 3002 1300 -1
Total = 6352

Enter production data for plant number 3.
Enter number of units produced by each department.
Append a negative number to the end of the list.
5000 4020 500 4348 -1
Total = 13868

Enter production data for plant number 4.
Enter number of units produced by each department.
Append a negative number to the end of the list.
2507 6050 1809 -1
Total = 10366

Units produced in thousands of units: Plant #1 ******
Plant #2 ******
Plant #3 **************
Plant #4 **********
```

√ 自测题

13. 为函数 oneMore 写函数定义，它获取一个整数数组，函数使每个数组元素的值递增 1。自行添加需要的其他参数。

14. 给定以下函数定义：

```
void too2(int a[], int howMany)
{
```

```
    for (int index = 0; index < howMany; index++)
        a[index] = 2;
}
```

以下哪些函数调用是可以接受的?

```
int myArray[29];
too2(myArray, 29);
too2(myArray, 10);
too2(myArray, 55);
"Hey too2. Please, come over here."
int yourArray[100];
too2(yourArray, 100);
too2(myArray[3], 29);
```

15. 在可以更改为常量数组参数的任何数组参数之前插入 const：

```
void output(double a[], int size);
// 前条件：a[0]～a[size - 1]已被赋值
// 后条件：输出 a[0]～a[size - 1]的值

void dropOdd(int a[], int size);
// 前条件：a[0]～a[size - 1]已被赋值
// 后条件：a[0]～a[size - 1]的所有奇数变成 0
```

16. 写名为 outOfOrder 的函数，一个参数是 double 类型的数组，另一个是名为 size 的 int 参数，返回 int 值。该函数测试数组是否乱序，换言之，数组是否违反以下规则：

```
a[0] <= a[1] <= a[2] <= …
```

如果所有元素都符合上述规则(没有乱序)，函数返回-1；否则返回开始乱序的第一个元素的索引。例如以下声明：

```
double a[10] = {1.2, 2.1, 3.3, 2.5, 4.5, 7.9, 5.4, 8.7, 9.9, 1.0};
```

在上述数组中，a[2]和 a[3]是第一个乱序对，而 a[3]是乱序的第一个元素，所以函数返回 3。如果所有数组元素都按升序排列，函数返回-1。

7.3 数组编程

你绝不可以依靠一般印象，而要重点关注细节。

——*阿瑟·柯南道尔，《福尔摩斯探案集·身份案》*

本节要讨论部分填充的数组，并简单介绍数组的排序与搜索。本节没有与 C++语言有关的新内容，主要是更多地练习 C++数组参数的使用。

部分填充数组

写程序时，数组准确长度经常未知，或要求每次运行都改变。为解决该问题，一个常见而且简单的办法是将数组声明成可能需要的最大长度。然后就可自由使用数组或多或少的一部分。

但使用部分填充的数组时要谨慎。程序必须跟踪了解数组使用了多大一部分，绝对不能引用尚未赋值的任何索引变量。图 7.9 的程序演示了这个问题。程序读入一个高尔夫比分列表，显示每个比分与平均分的差值。列表长度最短 1 个比分，最长 10 个比分。比分存储在数组 score 中，它有 10 个索引变量，但程序只使用自己真正需要的那一部分。变量

numberUsed 跟踪记录数组中存储了多少个元素。元素(也就是比分)的存储位置是 score[0]～score[numberUsed - 1]。

从表面看，程序好像是将 numberUsed 作为数组的声明长度，并使用整个数组(而不是使用其中一部分)。任何操纵部分填充数组的函数，通常必须将变量 numberUsed 作为参数传给它。如使用得当，参数 numberUsed 能确保函数不会引用非法数组索引，所以有时能(但并非总能)避免用参数给出数组的声明长度。例如，函数 showDifference 和 computeAverage 就用参数 numberUsed 来确保只使用合法数组索引。但函数 fillArray 必须知道数组的最大声明长度，确保不会在数组中填充过多的内容。

图 7.9　部分填充数组

```
1  // 显示每个高尔夫比分与平均比分的差值
2  #include <iostream>
3  const int MAX_NUMBER_SCORES = 10;

4  void fillArray(int a[], int size, int& numberUsed);
5  // 前条件：size 是数组 a 的声明长度
6  // 后条件：numberUsed 是数组 a 实际存储的值的数目
7  // a[0]～a[numberUsed - 1]使用从键盘读入的非负整数来填充

8  double computeAverage(const int a[], int numberUsed);
9  // 前条件：a[0]～a[numberUsed - 1]已被赋值；numberUsed > 0
10 // 返回 a[0]～ a[numberUsed - 1]的平均值

11 void showDifference(const int a[], int numberUsed);
12 // 前条件：前面的 numberUsed 个索引变量已被赋值
13 // 后条件：针对前面的 numberUsed 个元素，输出每个元素与平均值的差值

14 int main()
15 {
16     using namespace std;
17     int score[MAX_NUMBER_SCORES], numberUsed;

18     cout << "This program reads golf scores and shows\n"
19          << "how much each differs from the average.\n";
20
21     cout << "Enter golf scores:\n";
22     fillArray(score, MAX_NUMBER_SCORES, numberUsed);
23     showDifference(score, numberUsed);

24     return 0;
25 }

26 // 使用 iostream:
27 void fillArray(int a[], int size, int& numberUsed)
28 {
29     using namespace std;
30     cout << "Enter up to " << size << " nonnegative whole numbers.\n"
31          << "Mark the end of the list with a negative number.\n";
32     int next, index = 0;
33     cin >> next;
34     while ((next >= 0) && (index < size))
35     {
36         a[index] = next;
37         index++;
38         cin >> next;
39     }

40     numberUsed = index;
```

```
41  }
42  double computeAverage(const int a[], int numberUsed)
43  {
44      double total = 0;
45      for (int index = 0; index < numberUsed; index++)
46          total = total + a[index];
47      if (numberUsed > 0)
48      {
49          return (total/numberUsed);
50      }
51      else
52      {
53          using namespace std;
54          cout << "ERROR: number of elements is 0 in computeAverage.\n"
55               << "computeAverage returns 0.\n";
56          return 0;
57      }
58  }
59  void showDifference(const int a[], int numberUsed)
60  {
61      using namespace std;
62      double average = computeAverage(a, numberUsed);
63      cout << "Average of the " << numberUsed
64           << " scores = " << average << endl
65           << "The scores are:\n";
66      for (int index = 0; index < numberUsed; index++)
67      cout << a[index] << " differs from average by "
68           << (a[index] - average) << endl;
69  }
```

示范对话

```
This program reads golf scores and shows
how much each differs from the average.
Enter golf scores:
Enter up to 10 nonnegative whole numbers.
Mark the end of the list with a negative number.
69 74 68 -1

Average of the 3 scores = 70.3333
The scores are:
69 differs from average by -1.33333
74 differs from average by 3.66667
68 differs from average by -2.33333
```

编程提示：不要吝啬形参

注意图 7.9 的 `fillArray` 函数。调用时将声明数组长度 MAX_ NUMBER_SCORES 作为实参之一，例如：

```
fillArray(score, MAX_NUMBER_SCORES, numberUsed);
```

你可能认为，由于 MAX_NUMBER_SCORES 是全局定义常量，所以能直接在 `fillArray` 的定义中使用，不需要把它作为参数传递。也许正确；如 `fillArray` 不在除图 7.9 之外的其他任何程序中使用，完全可以不把 MAX_NUMBER_SCORES 当作 `fillArray` 的参数。但 `fillArray` 是很有用的常规函数，可在许多不同的程序中使用。事实上，下一小节讨论的

程序(图 7.10)会再次用到 fillArray 函数。届时，用来表示声明数组长度的参数是一个不同名称的全局常量。将全局变量 MAX_NUMBER_SCORES 集成到 fillArray 函数主体中，就无法在图 7.10 中重用该函数。

即使 fillArray 只在一个程序中使用，仍有必要将数组的声明长度作为 fillArray 的实参。这样可提醒自己注意该实参对于函数的重要性。

编程实例　搜索数组

一个常见编程任务是在数组中搜索给定值。例如，数组可能包含修一门课的所有学生的学生编号。要知道一个特定的学生是否注册，就可搜索该数组，判断其中是否包含那个学生的编号。图 7.10 的程序首先填充一个数组，然后搜索由用户指定的值。一个真正的应用程序应该更全面，但本例确实展示了顺序搜索算法的所有基本原理。**顺序搜索**是最容易想到的搜索算法，它从头到尾检查所有数组元素，判断每个数组元素是否等于目标值。

图 7.10 用 search 函数搜索数组。搜索时，不仅想知道目标值是否在数组中，而且想知道容纳着那个目标值的索引变量的索引，因为索引也许能传达有关目标值的更多的信息。所以，只要目标值在数组中，就让 search 函数返回一个索引，给出目标值在数组中的位置。如果目标值不在数组中，search 返回-1。下面深入分析 search 函数。

search 函数用 while 循环逐个检查数组元素，判断每个元素是否等于目标值。变量 found 用作一个标志，记录是否发现了目标元素。发现目标元素，found 会被设为 true，而这会终止 while 循环。

图 7.10　搜索数组

```
1  // 搜索用非负整数来部分填充的数组
2  #include <iostream>
3  const int DECLARED_SIZE = 20;

4  void fillArray(int a[], int size, int& numberUsed);
5  // 前条件：size 是数组 a 的声明长度
6  // 后条件：numberUsed 是数组 a 实际存储的值的数目
7  // a[0]~a[numberUsed-1]使用从键盘读入的非负整数来填充

8  int search(const int a[], int numberUsed, int target);
9  // 前条件：numberUsed <= a 的声明长度
10 // 另外，a[0]~a[numberUsed -1]已被赋值
11 // 返回 a[index] == target 的第一个索引，前提是有这样的一个索引，否则返回-1

12 int main()
13 {
14    using namespace std;
15    int arr[DECLARED_SIZE], listSize, target;

16    fillArray(arr, DECLARED_SIZE, listSize);

17    char ans;
18    int result;
19    do
20    {
21       cout << "Enter a number to search for: ";
22       cin >> target;

23       result = search(arr, listSize, target);
24       if (result == -1)
25          cout << target << " is not on the list.\n";
26       else
27          cout << target << " is stored in array position "
28               << result << endl
29               << "(Remember: The first position is 0.)\n";
```

```
30          cout << "Search again?(y/n followed by Return): ";
31          cin >> ans;
32      } while ((ans != 'n') && (ans != 'N'));

33      cout << "End of program.\n";
34      return 0;
35  }
36  // 使用 iostream:
37  void fillArray(int a[], int size, int& numberUsed)

    <fillArray 函数其余定义在图 7.9 中给出>

38  int search(const int a[], int numberUsed, int target)
39  {
40      int index = 0;
41
42      bool found = false;
43      while ((!found) && (index < numberUsed))
44          if (target == a[index])
45              found = true;
46          else
47              index++;
48
49      if (found)
50          return index;
51      else
52          return -1;
53  }
```

示范对话

```
Enter up to 20 nonnegative whole numbers.
Mark the end of the list with a negative number.
10 20 30 40 50 60 70 80 -1
Enter a number to search for: 10
10 is stored in array position 0.
(Remember: The first position is 0.)
Search again?(y/n followed by Return): y
Enter a number to search for: 40
40 is stored in array position 3.
(Remember: The first position is 0.)
Search again?(y/n followed by Return): y
Enter a number to search for: 42
42 is not on the list.
Search again?(y/n followed by Return): n
End of program.
```

编程实例　数组排序

一个常见编程任务是对一组值进行排序，人们对这个问题进行了非常全面的研究。例如，一组销售数据可能要求从最小值到最大值排列，或要求从最大值到最小值排列；而一组英文单词可能要求按字母顺序排列。本节介绍 sort 函数，它对一个部分填充数组中的数字进行排序，使它们从小到大排列。

sort 函数接收数组形参 a。数组被部分填充，所以还有一个名为 numberUsed 的形参指出实际使用了数组中的多少个位置。因此，sort 函数的声明和前条件如下：

```
void sort(int a[], int numberUsed);
// 前条件：numberUsed <= 数组 a 的声明长度
// 数组元素 a[0]～a[numberUsed - 1]已被赋值
```

sort 函数要重新排列数组 a 中的元素。函数调用结束后，元素要像下面这样排列：

```
a[0] ≤ a[1] ≤ a[2] ≤ . . . ≤ a[numberUsed - 1]
```

采用的排序算法称为“选择排序”，它是最容易理解的排序算法之一。

视频讲解：*Selection Sort Walkthrough*

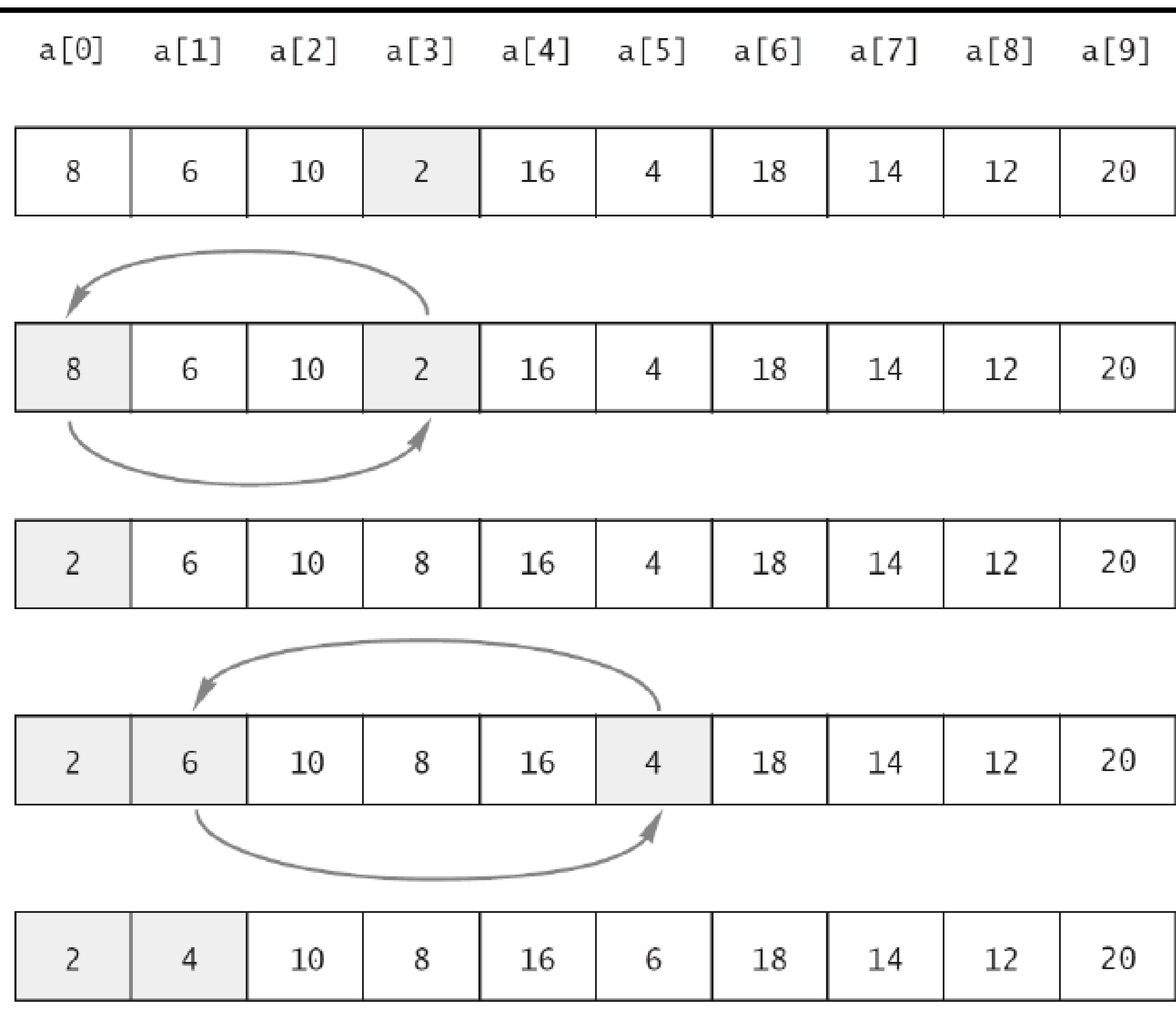
设计算法的一种方式是依赖问题定义。在本例中，问题是让数组 a 的各个元素从小到大排列。这意味着排序后 a[0]最小，a[1]倒数第二小，依此类推。根据它就能概括出**选择排序**算法的基本结构：

```
for (int index = 0; index < numberUsed; index++)
    将第 index 个最小的元素放到 a[index]中
```

有许多办法都能实现。一个是使用两个数组，按排序顺序将元素从一个数组复制到另一个数组。但一个数组应该足够，这样更经济。所以，我们的 sort 函数只使用一个数组，其中包含了准备排序的值。sort 函数通过交换值对的方式来重新排列数组 a 中的值。下面来分析一下具体的例子，以便理解算法的工作原理。

以图 7.11 显示的数组为例。算法准备将最小的值放到 a[0]中。刚开始，最小值在 a[3]中。因此，算法会交换 a[0]和 a[3]的值。然后，算法查找下个较小的元素。由于 a[0]的值已经最小，所以下一个较小的元素肯定是 a[1]，a[2]，a[3]，…，a[9]等剩余元素中最小的那个。在图 7.11 的例子中，下个最小元素是 a[5]，所以算法会交换 a[1]和 a[5]的值。这一定位和交换过程在图 7.11 的第 4 幅和第 5 幅数组图中反映出来。之后，算法会继续定位第三个较小的元素，依此类推。

图 7.11　选择排序



随着排序过程的继续，位于前面的数组元素被逐渐设置成正确的、排好序的值。换言之，数组中未排序的元素都位于数组的后面，而且它们的数目越来越少。注意，该算法不需要对最后一个索引变量 a[9]的值采取任何操作，因为一旦其他所有元素都正确定位，a[9]

肯定已经获得了正确的值。毕竟，a[9]的值应该是数组中最后要移动的值。但等到处理 a[9]的时候，它已经无处可移了，只能放在 a[9]中。

图 7.12 给出了 sort 函数的定义，并提供了演示程序。sort 首先用 indexOfSmallest 函数在数组"未排序部分"中查找最小元素的索引。然后，它执行一次交换，将这个元素移动到数组"已排序部分"的末端。

图 7.12 的 swapValues 函数用于交换两个索引变量的值。例如，以下调用会交换 a[0]和 a[3]的值：

```
swapValues(a[0], a[3]);
```

注意，第 5 章已解释了 swapValues 函数。

图 7.12　数组排序

```
1  // 测试排序过程
2  #include <iostream>
3  void fillArray(int a[], int size, int& numberUsed);
4  // 前条件：size 是数组 a 的声明长度
5  // 后条件：numberUsed 是数组 a 实际存储的值的数目
6  // a[0]~a[numberUsed - 1]使用从键盘读入的非负整数来填充
7  void sort(int a[], int numberUsed);
8  // 前条件：numberUsed <= 数组 a 的声明长度
9  // 数组元素 a[0]~a[numberUsed - 1]已被赋值
10 // 后条件：a[0]~a[numberUsed - 1]的值被重新排列
11 // 使 a[0] <= a[1] <= ... <= a[numberUsed - 1]
12 void swapValues(int& v1, int& v2);
13 // 交换 v1 和 v2 的值

14 int indexOfSmallest(const int a[], int startIndex, int numberUsed);
15 // 前条件：0 <= startIndex < numberUsed。引用的数组元素已被赋值
16 // 返回索引 i，使 a[i]是 a[startIndex], a[startIndex + 1], ..., a[numberUsed - 1]
17 // 等值中最小的

18 int main()
19 {
20     using namespace std;
21     cout << "This program sorts numbers from lowest to highest.\n";

23     int sampleArray[10], numberUsed;
24     fillArray(sampleArray, 10, numberUsed);
25     sort(sampleArray, numberUsed);

26     cout << "In sorted order the numbers are:\n";
27     for (int index = 0; index < numberUsed; index++)
28         cout << sampleArray[index] << " ";
29     cout << endl;

30     return 0;
31 }

32 // 使用 iostream:
33 void fillArray(int a[], int size, int& numberUsed)

     <fillArray 其余定义在图 7.9 中给出>
```

```
34 void sort(int a[], int numberUsed)
35 {
36     int indexOfNextSmallest;
37     for (int index = 0; index < numberUsed - 1; index++)
38     {// 将正确的值放在 a[index]中:
39         indexOfNextSmallest =
40                 indexOfSmallest(a, index, numberUsed);
41         swapValues(a[index], a[indexOfNextSmallest]);
42         // a[0] <= a[1] <=...<= a[index]都是原始数组元素中最小的
43         // 剩余的元素在剩余的位置
44     }
45 }
46 void swapValues(int& v1, int& v2)
47 {
48     int temp;
49     temp = v1;
50     v1 = v2;
51     v2 = temp;
52 }
53
54 int indexOfSmallest(const int a[], int startIndex, int numberUsed)
55 {
56     int  min = a[startIndex],
57          indexOfMin = startIndex;
58     for (int index = startIndex + 1; index < numberUsed; index++)
59         if (a[index] < min)
60         {
61             min = a[index];
62             indexOfMin = index;
63             // min 是 a[startIndex]~a[index]中最小的
64         }
65
66     return indexOfMin;
67 }
```

示范对话

```
This program sorts numbers from lowest to highest.
Enter up to 10 nonnegative whole numbers.
Mark the end of the list with a negative number.
80 30 50 70 60 90 20 30 40 -1
In sorted order the numbers are:
20 30 30 40 50 60 70 80 90
```

编程实例 冒泡排序

视频讲解：*Bubble Sort Walkthrough*

刚才描述的选择排序算法并非唯一排序方式。计算机科学家甚至为各种排序算法弄了一个排行榜。有的算法效率比其他高，有的则只适合特定数据类型。和选择排序一样，冒泡排序也是一种简单和常规的排序算法。

用冒泡排序对数组进行升序排序，较大值会如同气泡上浮，连续向数组末尾移动。假定对以下整数数组进行排序：

初始数组： {3, 10, 9, 2, 5}

第一轮将最大值 10 移到数组末尾：

第一轮的结果： {3, 9, 2, 5, 10}

第二轮将第二大值 9 移至数组倒数第二个位置：

第二轮的结果： {3, 2, 5, 9, 10}

第三轮将第三大值 5 移至数组倒数第三个位置(就是当前位置)：

第三轮的结果： {3, 2, 5, 9, 10}

第四轮将第四大值 3 移至数组倒数第四个位置：

第四轮的结果： {2, 3, 5, 9, 10}

这时算法结束。数组开头的数字不用检查，因为唯一剩下的数字肯定最小。设计基于冒泡排序的程序时，注意最大值要放到索引位置 length-1，第二大放到 length-2，以此类推。因此，对应的循环应该从索引 length-1 开始，倒数至索引 1。索引 0 不需要考虑，因为它肯定包含最小值。可用以下代码实现该循环，变量 i 是目标索引：

```
for (int i = length-1; i > 0; i--)
```

“冒泡”过程在循环的每次迭代中发生。这要求用一个嵌套循环将最大值移至数组索引 i 处。首先，索引 0 到索引 i 之间最大的值冒泡到索引 i。为此，第一步是将索引 0 的值和索引 1 的位置进行比较。前者大于后者就交换两个值，结果是索引 1 具有较大的值。反之，如果前者小于或等于后者，则什么都不发生。对于以下未排序的数组：

初始数组：　　{3, 10, 9, 2, 5}

上述步骤造成 3 和 10 比较。由于 10 > 3，所以什么都不发生，10 留在索引 1：

步骤 1 的结果： {3, 10, 9, 2, 5}

要一直重复这个过程，直到抵达索引 i.。第二步是比较索引 1 和索引 2 的值，造成两个值进行交换：

步骤 2 的结果： {3, 9, 10, 2, 5}

如此再重复两次：

步骤 3 的结果： {3, 9, 2, 10, 5}

步骤 4 的结果： {3, 9, 2, 5, 10}

这样就结束了冒泡排序第一轮的处理，整个数组中的最大值成功冒泡到数组末尾。下一轮是对第二大的值进行冒泡，并依此类推。在外层循环中，变量 i 始终代表要冒泡到的目标索引。假定用变量 j 代表正在冒泡的项的索引，那么内外层循环如下所示：

```
for (int i = length-1; i > 0; i--)
  for (int j = 0; j < i; j++)
```

内层循环比较索引 j 和 j+1 的值。较大的移至(或保留在)索引 j+1。下面展示了完整算法，图 7.13 是一个完整的程序。

```
for (int i = length-1; i > 0; i--)
  for (int j = 0; j < i; j++)
    if (arr[j] > arr[j+1])
    {
      int temp = arr[j+1];
```

```
            arr[j+1] = arr[j];
            arr[j] = temp;
        }
```

图 7.13　冒泡排序程序

```
1  // 图 7.13 冒泡排序程序
2  // 使用冒泡排序对一个整数数组进行排序
3  #include <iostream>
4
5  void bubblesort(int arr[], int length);
6  // 前条件: length <= 数组 arr 的声明长度
7  // 数组元素 arr[0]到 a[length - 1]均已赋值
8  // 后条件: arr[0]到 arr[length - 1]被重新排序:
9  // arr[0] <= a[1] <= <= arr[length - 1].
10
11 int main()
12 {
13    using namespace std;
14    int a[] = {3, 10, 9, 2, 5, 1};
15
16    bubblesort(a, 6);
17    for (int i=0; i<6; i++)
18    {
19       cout << a[i] << " ";
20    }
21    cout << endl;
22    return 0;
23 }
24
25 void bubblesort(int arr[], int length)
26 {
27    // 较大的数字向右边移动
28    for (int i = length-1; i > 0; i--)
29       for (int j = 0; j < i; j++)
30          if (arr[j] > arr[j+1])
31          {
32             // 交换数字
33             int temp = arr[j+1];
34             arr[j+1] = arr[j];
35             arr[j] = temp;
36          }
37 }
```

示范对话

```
1 2 3 5 9 10
```

√ 自测题

17. 写程序将最多 10 个非负整数读入 numberArray 数组，在屏幕上回显这些整数。做这道题不需要使用任何函数。这只是练习程序，应尽可能地小。

18. 写程序将最多 10 个字母读入数组，按相反顺序在屏幕上回显这些字母。例如，假定输入如下：

    ```
    abcd.
    ```

    那么输出如下：

    ```
    dcba
    ```

    输入时，最后的句点符号必不可少，它是标记输入结束的哨兵值。将数组命名为 letterBox。做这道题不需要使用任何函数。这只是练习程序，应尽可能地小。

19. 下面是图 7.12 定义的 search 函数的修订版本。要使用 search 函数的这个修订版本，需要稍微修改一下程序。但就本题来说，你唯一要做的就是为这个版本的 search 写函数定义。

```
bool search(const int a[], int numberUsed, int target, int& where);
// 前条件：numberUsed <= 数组 a 的声明长度；a[0]~a[numberUsed -1]已被赋值
// 后条件：如果 a[0]~a[numberUsed - 1]的某个元素等于 target，该函数返回 true，
// 并设置 where 的值，使 a[where] == target；
// 否则，该函数返回 false，而且不修改 where 的值
```

7.4 多 维 数 组

索引两个比一个好。

——*某计算机系厕所墙壁*

C++允许声明具有多个索引的数组。本节介绍这种多维数组。

多维数组基础

有时要让一个数组具有多个索引，C++提供了这方面的支持。以下语句声明名为 page 的字符数组。该数组有两个索引：第一个索引范围是 0～29；第二个索引范围是 0～99。

```
char page[30][100];
```

该数组的每个索引变量都有两个索引值。例如，page[0][0]，page[15][32]和 page[29][99]就是该数组的三个索引变量。注意每个索引都必须单独放到方括号中。和一维数组相同，多维数组的每个索引变量其实都是数组基类型的一个变量。

数组可以有任意数量的索引，但最常见的索引数量是两个，这种数组称为二维数组。二维数组可转换成对应的二维显示。其中，第一个索引代表行，第二个索引代表列。例如，二维数组 page 的索引变量可以像下面这样排列：

```
page[0][0], page[0][1], ……, page[0][99]
page[1][0], page[1][1], ……, page[1][99]
page[2][0], page[2][1], ……, page[2][99]
⋮
page[29][0], page[29][1], ……, page[29][99]
```

例如，可用 page 数组存储一个书页上的所有字符。这一页总共有 30 行(编号 0～29)，每行 100 个字符(编号 0～99)。

在 C++中，像 page 这样的二维数组，实际是由数组构成的数组。page 数组实际是长度为 30 的一维数组，它的基类型是长度为 100 的一维字符数组。但是，通常都不必关心这一点，可放心地认为 page 数组是有两个索引的数组(而不要认为它是由数组构成的数组，因为那较难跟踪和维护)。但至少在一种情况下，二维数组看起来更像是一个由数组构成的数组。在这种情况下，你定义的函数将使用一个二维数组参数。详情在下一小节讨论。

多维数组声明

语法

Type arrayName[Size_Dim_1] [Size_Dim_2]. . .[Size_Dim_Last];

示例

```
char page[30][100];
int matrix[2][3];
double threeDPicture[10][20][30];
```

以上形式的数组声明会为数组索引的每个组合都定义索引变量。例如，对于上述第二个示范声明，会为 matrix 数组定义 6 个索引变量：

```
matrix[0][0], matrix[0][1], matrix[0][2],
matrix[1][0], matrix[1][1], matrix[1][2]
```

多维数组参数

以下二维数组声明实际声明长度为 30 的一维数组，数组基类型是长度为 100 的一维字符数组：

```
char page[30][100];
```

将二维数组视为由数组构成的数组，有助于理解 C++如何处理作为参数使用的多维数组。

例如，以下函数获取一个数组实参(比如 page)，将它打印到屏幕：

```
void displayPage(const char p[][100], int sizeDimension1)
{
    for (int index1 = 0; index1 < sizeDimension1; index1++)
    { // 打印一行：
        for (int index2 = 0; index2 < 100; index2++)
            cout << p[index1][index2];
        cout << endl;
    }
}
```

注意，二维数组作为参数传递时不指定第一维的长度，所以必须包括一个 int 参数来给出第一维的长度(和普通数组一样，编译器允许在第一对方括号中添加数字来指定第一维的长度。但这样的一个数字只能充当注释；编译器会忽略任何这样的数字)。第二维的长度在数组参数之后给出(如果不止二维，其他维的长度也要给出)，例如：

```
const char p[][100]
```

如果明白多维数组实际是数组构成的数组，就很容易理解这一规则。对于以下二维数组参数：

```
const char p[][100]
```

由于 p 是数组构成的数组，所以第一维才是 p 数组的真正索引，会被当作普通一维数组的数组索引进行处理。第二维(和更多维)是基类型描述的一部分，指出基类型是长度为 100 的字符数组。

多维数组参数

在函数头或者函数声明中使用多维数组参数时，第一维的长度不要给出，但剩余的其他维的长度必须用单独的方括号给出。由于第一维长度没有给出，所以通常要用另一个 int 参数给出第一维的长度。下面是使用了二维数组参数 p 的示范函数声明：

```
void getPage(char p[][100], int sizeDimension1);
```

编程实例　二维打分程序

图 7.14 的程序使用名为 grade 的二维数组存储并显示一个小班的成绩。班上有 4 名学生，要参加 3 门测验。

图 7.14　二维数组

```
1   // 将每个学生的测验分数读入二维数组 grade 中(但本例不显示负责输入的代码)
2   // 计算每个学生的平均分数，以及每门测验的平均分数。显示测验分数和平均分
3   #include <iostream>
4   #include <iomanip>
5   const int NUMBER_STUDENTS = 4, NUMBER_QUIZZES = 3;
6
7   void computeStAve(const int grade[][NUMBER_QUIZZES], double stAve[]);
8   // 前条件：全局变量 NUMBER_STUDENTS 和 NUMBER_QUIZZES 是 grade 数组的每一维的长度
9   // 每个索引变量 grade[stNum-1, quizNum-1]都包含学生 stNum 所参加的测验 quizNum 的分数
10  // 后条件：每个 stAve[stNum-1]都包含学生 stNum 的平均分数
11
12  void computeQuizAve(const int grade[][NUMBER_QUIZZES], double quizAve[]);
13  // 前条件：全局变量 NUMBER_STUDENTS 和 NUMBER_QUIZZES 是 grade 数组的每一维的长度
14  // 每个索引变量 grade[stNum-1, quizNum-1] 都包含学生 stNum 所参加的测验 quizNum 的分数
15  // 后条件：每个 quizAve[quizNum-1]都包含测验 quizNum 的平均分数
16
17  void display(   const int grade[][NUMBER_QUIZZES],
18                  const double stAve[], const double quizAve[]);
19  // 前条件：全局常量 NUMBER_STUDENTS 和 NUMBER_QUIZZES 是 grade 数组的每一维的长度
20  // 每个索引变量 grade[stNum-1, quizNum-1] 都包含学生 stNum 所参加的测验 quizNum 的分数
21  // 每个 stAve[stNum-1] 都包含学生 stNum 的平均分数
22  // 每个 quizAve[quizNum-1]都包含测验 quizNum 的平均分数
23  // 后条件：输出 grade，stAve 和 quizAve 中的所有数据
24
25  int main()
26  {
27      using namespace std;
28      int grade[NUMBER_STUDENTS][NUMBER_QUIZZES];
29      double stAve[NUMBER_STUDENTS];
30      double quizAve[NUMBER_QUIZZES];
31

    <这里是用于填充 grade 数组的代码，这里省略>

32      computeStAve(grade, stAve);
33      computeQuizAve(grade, quizAve);
34      display(grade, stAve, quizAve);
35      return 0;
36  }

37  void computeStAve(const int grade[][NUMBER_QUIZZES], double stAve[])
38  {
39      for (int stNum = 1; stNum <= NUMBER_STUDENTS; stNum++)
```

```
40      {// 处理 1 个 stNum:
41          double sum = 0;
42          for (int quizNum = 1; quizNum <= NUMBER_QUIZZES; quizNum++)
43              sum = sum + grade[stNum-1][quizNum-1];
44          // sum 包含学生 stNum 的所有测验分数的总和
45          stAve[stNum-1] = sum / NUMBER_QUIZZES;
46          // 学生 stNum 的平均分数就是 stAve[stNum-1]的值
47      }
48  }
49
50  void computeQuizAve(const int grade[][NUMBER_QUIZZES], double quizAve[])
51  {
52      for (int quizNum = 1; quizNum <= NUMBER_QUIZZES; quizNum++)
53      {// 处理一门测验(针对所有学生):
54          double sum = 0;
55          for (int stNum = 1; stNum <= NUMBER_STUDENTS; stNum++)
56              sum = sum + grade[stNum-1][quizNum-1];
57          // sum 包含参加测验 quizNum 的所有学生的分数之和
58          quizAve[quizNum-1] = sum / NUMBER_STUDENTS;
59          // 测验 quizNum 的平均分数就是 quizAve[quizNum-1]的值
60      }
61  }
62
63  // 使用 iostream 和 iomanip:
64  void display(const int grade[][NUMBER_QUIZZES],
65                      const double stAve[], const double quizAve[])
66  {
67      using namespace std;
68      cout.setf(ios::fixed);
69      cout.setf(ios::showpoint);
70      cout.precision(1);
71      cout << setw(10) << "Student"
72           << setw(5) << "Ave"
73           << setw(15) << "Quizzes\n";
74      for (int stNum = 1; stNum <= NUMBER_STUDENTS; stNum++)
75      {// 针对一个 stNum 的显示:
76          cout << setw(10) << stNum
77               << setw(5) << stAve[stNum-1] << " ";
78          for (int quizNum = 1; quizNum <= NUMBER_QUIZZES; quizNum++)
79              cout << setw(5) << grade[stNum-1][quizNum-1];
80          cout << endl;
81      }
82      cout << "Quiz averages = ";
83      for (int quizNum = 1; quizNum <= NUMBER_QUIZZES; quizNum++)
84          cout << setw(5) << quizAve[quizNum-1];
85      cout << endl;
86  }
```

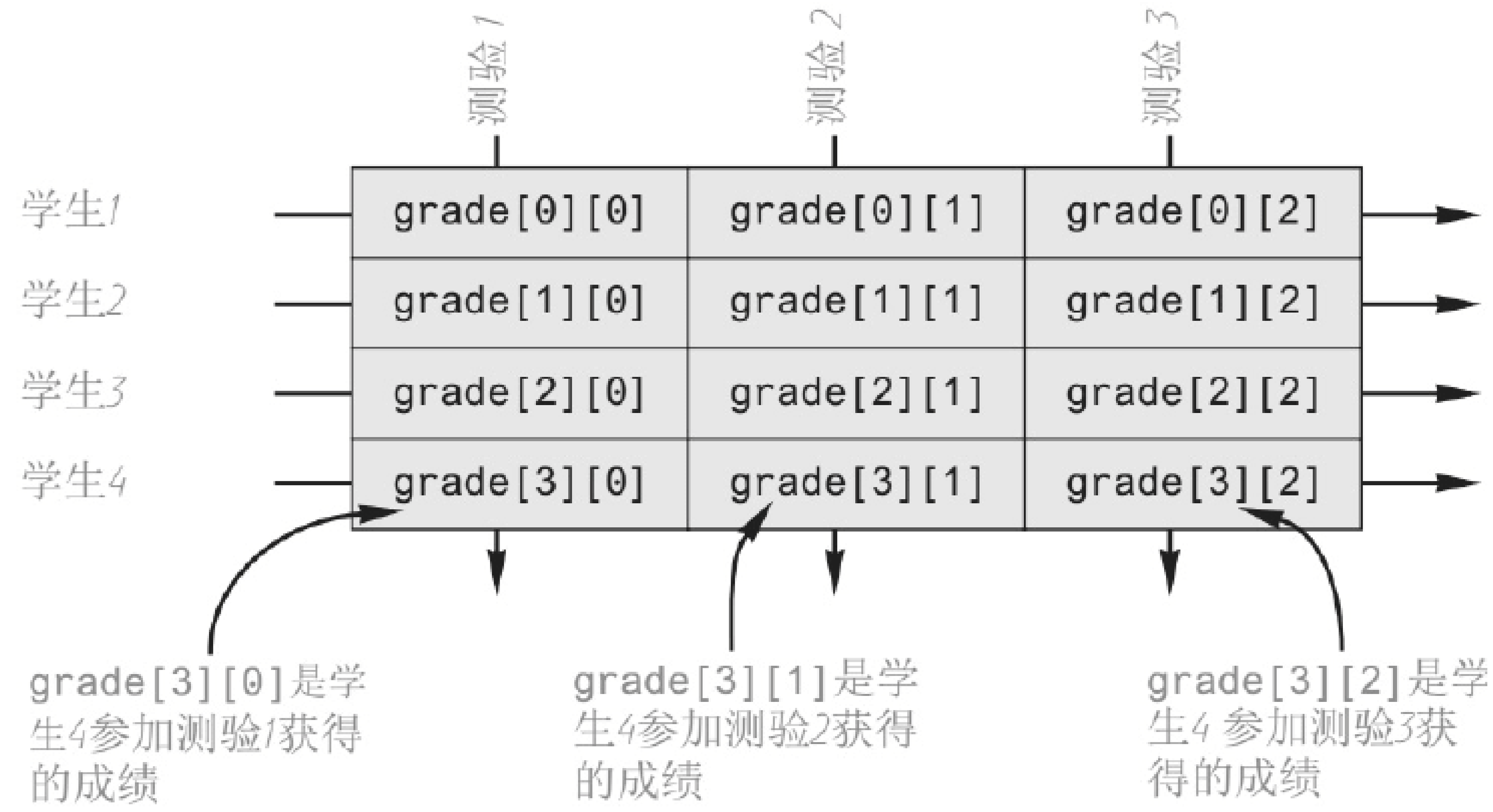
示范对话

<填充 grade 数组的对话省略>

```
Student        Ave              Quizzes
1             10.0        10       10       10
2              1.0         2        0        1
3              7.7         8        6        9
4              7.3         8        4       10
Quiz averages =          7.0      5.0      7.5
```

图 7.15 展示了如何用 grade 数组存储数据。第一个索引标识学生，第二个索引标识测验。由于学生和测验的编号都从 1(而非 0)开始，所以必须使学生和测验的编号减 1，才能得到存储特定测验分数的索引变量。例如，学生 4 的测验 1 的分数记录在 grade[3][0]中。

图 7.15　二维数组 grade



程序还使用了两个普通一维数组。其中，stAve 数组记录每个学生的平均测验分数。例如，程序将 stAve[0]设为学生 1 参加的所有测验的平均分数，将 stAve[1]设为学生 2 参加的所有测验的平均分数，依此类推。quizAve 数组记录每一门测验的平均分数。例如，程序将 quizAve[0]设为所有学生参加测验 1 所获得的平均分数，将 quizAve[1]设为所有学生参加测验 2 所获得的平均分数，依此类推。图 7.15 展示了 grade，stAve 和 quizAve 数组的关系。这幅图包含 grade 数组的一些示范数据，它们决定了程序存储在 stAve 和 quizAve 中的值。图 7.16 也显示了这些值，它们是程序为 stAve 和 quizAve 计算出来的。

图 7.16　二维数组 grade(另一个视图)

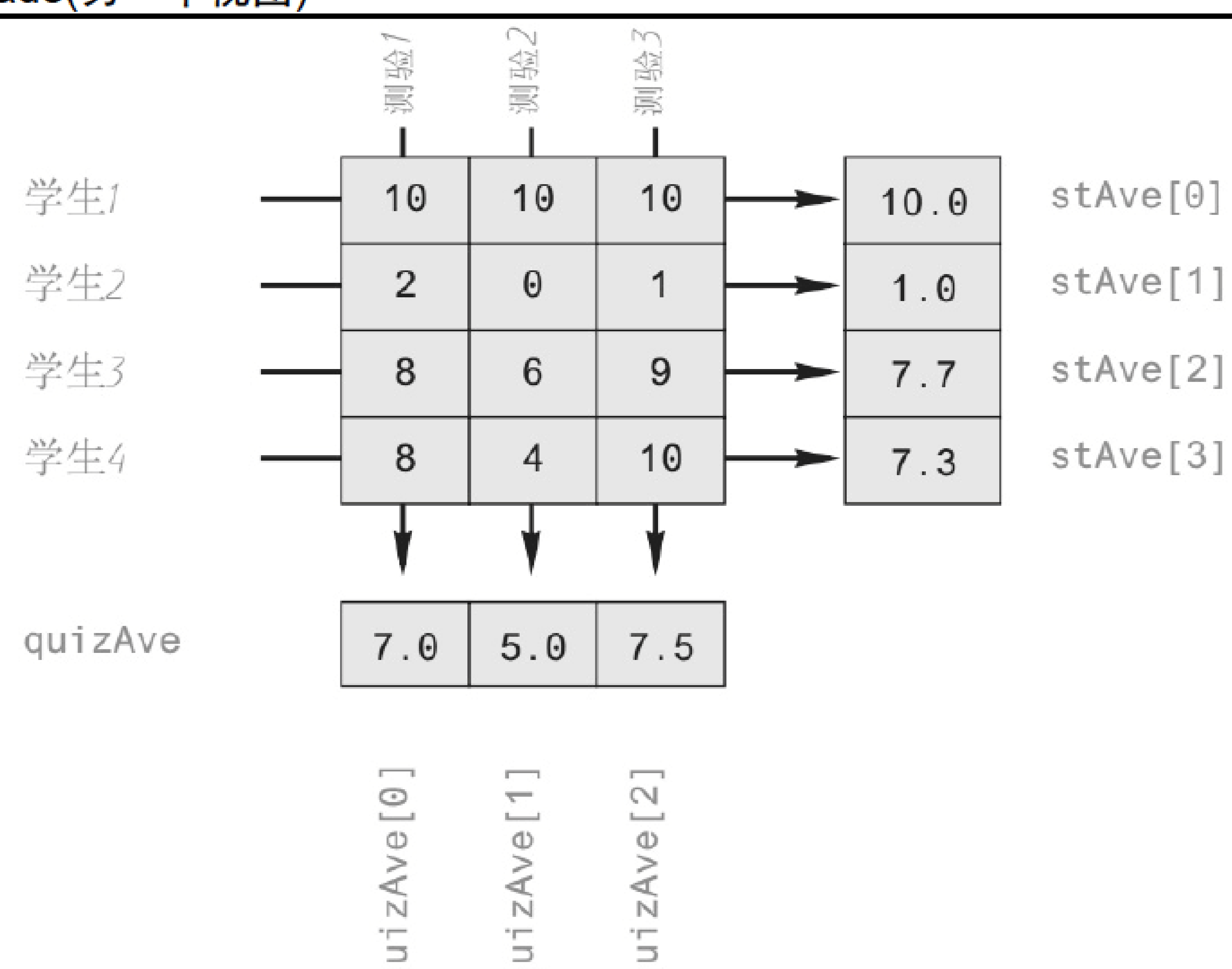


图 7.14 的完整程序填充 grade 数组，计算并显示学生平均分数与测验平均分数。在程序中，数组每一维的长度被声明为全局命名常量。由于其中定义的函数是这个程序特有的，不准备在其他地方重用，所以直接在函数主体中使用全局常量，而不是将数组每一维的长度作为参数传递。由于过于简单，书中没有列出用于填充数组的代码。

陷阱：在数组索引之间使用逗号

注意，图 7.13 将二维数组 grade 的索引变量写成 grade[stNum-1][quizNum-1]，其中使用了两对方括号。在另一些编程语言中，可能要求只用一对方括号，并用逗号加以分隔，比如 grade[stNum-1, quizNum-1]。在 C++中，这样写是不允许的。如果在 C++中使用 grade[stNum-1, quizNum-1]，也许不会产生任何错误消息，但由于用法不正确，所以可能导致程序表现异常。■

√ 自测题

20. 以下代码输出什么？

```
int myArray[4][4], index1, index2;
for(index1 = 0; index1 < 4; index1++)
    for (index2 = 0; index2 < 4; index2++)
        myArray[index1][index2] = index2;
for(index1 = 0; index1 < 4; index1++)
{
    for (index2 = 0; index2 < 4; index2++)
        cout << myArray[index1][index2] << " ";
    cout << endl;
}
```

21. 编写代码，在数组 a(声明见下)中填充从键盘输入的值。要求每行输入 5 个数，共 4 行(虽然你的方案不一定要依赖于在输入时对数字进行分行)。

```
int a[4][5]
```

22. 为名为 echo 的 void 函数写一个定义，要求以下函数调用能回显自测题 21 的输入，而且使用与输入时相同的格式来回显(也就是共 4 行，每行 5 个数)：

```
echo(a, 4);
```

小　结

- 数组可用于存储和操纵一个数据集合，其中的所有数据都具有相同的类型。
- 使用数组的索引变量与使用数组基类型的其他任何变量没有区别。
- for 循环是遍历数组元素并对每个索引变量都采取一些操作的好方式。
- 使用数组时，最易犯的错误是企图访问不存在的数据索引。必须检查数组处理循环的第一次和最后一次迭代，确它没有使用超出合法边界的索引。
- 数组参数(数组形参)既不是传值参数，也不是传引用参数，而是一种新的参数类型。数组形参与传引用参数的共同点在于，函数体中对形参的任何更改都会在函数调用时作用于传给函数的数组实参。
- 数组的索引变量在计算机内存中一个接一个地存储，所以整个数组会占用一个连续的内存存储区域。数组作为参数传给函数时，只有第一个索引变量(索引编号 0)的地址才会传给调用函数。所以，使用数组形参的函数通常还需要另一个 int 类型的形参来给出数组的长度。
- 使用部分填充的数组时，程序需要用另一个 int 类型的变量来跟踪了解实际使用了数组中的多少个存储位置。
- 为了让编译器知道一个数组实参不应该由函数更改，可在与那个实参对应的数组形参前插入修饰符 const。用 const 修饰的数组参数称为常量数组参数。
- 如果希望数组使用多个索引，可考虑使用多维数组，它实际是由数组构成的数组。

自测题答案

1. 语句 int a[5];是一个声明，其中的 5 是数组元素的数目。表达式 a[4]访问该数组中的一个存储位置(元素)。它访问的元素索引编号是 4，即数组的第 5 个(也是最后一个)元素。

2. a. score
   b. double
   c. 5
   d. 0～4
   e. score[0]，score[1]，score[2]，score[3]和 score[4]之一

3. a. 多了一个初始值。
   b. 正确。数组长度是 4。
   c. 正确。数组长度是 4。

4. abc

5.
```
1.1 2.2 3.3
1.1 3.3 3.3
```
   (记住，索引从 0 开始，而不是从 1 开始)

6.
```
0 2 4 6 8 10 12 14 16 18
0 4 8 12 16
```

7. sampleArray 的索引变量是 sampleArray[0]～sampleArray[9]，但这段代码试图填充 sampleArray[1]～sampleArray[10]。sampleArray[10]的索引 10 越界。

8. 有一个索引越界。index 等于 9 时，index + 1 等于 10，所以 a[index + 1]相当于 a[10]，它的索引是非法的。循环时应该少迭代一次。为了改正代码，只需将 for 循环的第一行改为：
```
for (int index = 0; index < 9; index++)
```

9.
```
int i, a[20];
cout << "Enter 20 numbers:\n";
for (i = 0; i < 20; i++)
   cin >> a[i];
```

10. 数组占用 14 个字节的内存。索引变量 yourArray[3]的地址是 1006。

11. 以下函数调用可以接受：
```
tripler(number);
tripler(a[2]);
tripler(a[number]);
```
   以下函数调用不正确：
```
tripler(a[3]);
tripler(a);
```
   第一个具有非法索引。第二个根本没有索引表达式。你不能将整个数组作为参数传给 tripler，就像上面的第二个非法调用那样。7.2 节讨论了如何将整个数组作为参数使用。

12. 循环会遍历索引变量 b[1]～b[5]，但 5 是数组 b 的非法索引。合法索引是 0，1，2，3 和 4。正确代码如下所示：
```
int b[5] = {1, 2, 3, 4, 5};
   for (int i = 0; i < 5; i++)
      tripler(b[i]);
```

13.
```
void oneMore(int a[], int size)
// 前条件：size 是数组 a 的声明长度。a[0]～a[size-1]已经赋值
// 后条件：针对 a 的所有索引变量，a[index]的值递增 1
{
```

```
    for (int index = 0; index < size; index++)
        a[index] = a[index] +1;
}
```

14. 以下函数调用可以接受：

```
too2(myArray, 29);
too2(myArray, 10);
too2(yourArray, 100);
```

以下调用虽然合法：

```
too2(myArray, 10);
```

但只会填充 myArray 的前 10 个索引变量。如果那正是你希望的，调用就可以接受。
以下函数调用不正确：

```
too2(myArray, 55);
"Hey too2. Please, come over here."
too2(myArray[3], 29);
```

第一个不正确，因为第二个参数太大。第二个不正确，因为它缺少最后的分号(以及其他原因)。第三个不正确，是因为它在本应使用整个数组的参数位置使用了一个索引变量。

15. 可使数组参数 output 成为一个常量参数，因为不需要更改数组参数的任何索引变量的值。但 dropOdd 中的参数不能成为常量参数，因为它的部分索引变量的值可能发生改变。

```
void output(const double a[], int size);
// 前条件：a[0]～a[size - 1]已被赋值
// 后条件：输出 a[0]～a[size - 1]的值

void dropOdd(int a[], int size);
// 前条件：a[0]～a[size - 1]已被赋值
// 后条件：a[0]～a[size - 1]的所有奇数变成 0
```

16.
```
int outOfOrder(double array[], int size)
{
    for (int i = 0; i < size - 1; i++)
        if (array[i] > array[i+1]) // 为每个 i 都获取 a[i + 1]
            return i+1;
    return -1;
}
```

17.
```
#include <iostream>
using namespace std;
const int DECLARED_SIZE = 10;

int main()
{
    cout << "Enter up to ten nonnegative integers.\n"
         << "Place a negative number at the end.\n";
    int numberArray[DECLARED_SIZE], next, index = 0;
    cin >> next;
    while ( (next >= 0) && (index < DECLARED_SIZE) )
    {
        numberArray[index] = next;
        index++;
        cin >> next;
    }
    int numberUsed = index;
    cout << "Here they are back at you: ";
    for (index = 0; index < numberUsed; index++)
        cout << numberArray[index] << " ";
    cout << endl;
    return 0;
}
```

18.
```
#include <iostream>
using namespace std;
const int DECLARED_SIZE = 10;
```

```
int main()
{
   cout << "Enter up to ten letters"
        << " followed by a period:\n";
   char letterBox[DECLARED_SIZE], next;
   int index = 0;
   cin >> next;
   while ( (next != '.') && (index < DECLARED_SIZE) )
   {
      letterBox[index] = next;
      index++;
      cin >> next;
   }

   int numberUsed = index;
   cout << "Here they are backwards:\n";
   for (index = numberUsed - 1; index >= 0; index--)
      cout << letterBox[index];
   cout << endl;
   return 0;
}
```

19.
```
bool search(const int a[], int numberUsed, int target, int& where)
{
   int index = 0;
   bool found = false;
   while ((!found) && (index < numberUsed))
   if (target == a[index])
      found = true;
   else
      index++;
   // 如果发现 target，那么 found == true，而且 a[index] == target
   if (found)
      where = index;
   return found;
}
```

20.
```
0 1 2 3
0 1 2 3
0 1 2 3
0 1 2 3
```

21.
```
int a[4][5];
int index1, index2;
for (index1 = 0; index1 < 4; index1++)
   for (index2 = 0; index2 < 5; index2++)
      cin >> a[index1][index2];
```

22.
```
void echo(const int a[][5], int sizeOfA)
// 输出数组 a 的值，共 sizeOfA 行，每行 5 个数字
{
   for (int index1 = 0; index1 < sizeOfA; index1++)
   {
      for (int index2 = 0; index2 < 5; index2++)
         cout << a[index1][index2] << " ";
      cout << endl;
   }
}
```

编 程 练 习

编程练习一般只需写很小的程序，运用本章提到的编程概念。

1. 写 firstLast2 函数获取一个整数数组和代表数组元素数量的整数。数组以 2 开头或结尾就返回 true，否则返回 false。用多个数组测试函数，要求这些数组具有不同长度，2 出现在数组开头和结尾，出现在中间，或者完全没有 2。

2. 写 countNum2s 函数获取一个整数数组和代表数组元素数量的整数。函数返回数组中 2 的个数。使用具有不同长度和 2 的个数的多个数组测试函数。

3. 写 swapFrontBack 函数获取一个整数数组和代表数组元素数量的整数。函数交换第一个和最后一个元素。要检查数组是否为空以防止出错。使用具有不同长度和前后数字的多个数组测试函数。

4. 以下代码创建一个小电话簿。一个数组存储姓名，另一个存储对应电话号码。例如，Michael Myers 的电话是 333-8000，Ash Williams 的电话是 333-2323。写 lookupName 函数正确查找并返回和指定姓名对应的电话号码。

```
int main()
{
   using namespace std;
   string names[] = {  "Michael Myers",
                       "Ash Williams",
                       "Jack Torrance",
                       "Freddy Krueger"};
   string phoneNumbers[] = {  "333-8000","333-2323",
                              "333-6150","339-7970"};
   string targetName, targetPhone;
   char c;
   do
   {
       cout << "Enter a name to find the "
            << "corresponding phone number."
            << endl;
        getline(cin, targetName);
        targetPhone = lookupName(targetName,
                         names, phoneNumbers,4);
        if (targetPhone.length() > 0)
           cout << "The number is: "
                << targetPhone << endl;
        else
           cout << "Name not found. "
                << endl;
        cout << "Look up another name? (y/n)"
             << endl;
        cin >> c;
        cin.ignore();
   } while (c == 'y');
   return 0;
}
```

编 程 项 目

编程项目要求综合运用多方面的知识来解决问题，程序一般比编程练习大，解题方式多样化。

编程项目 7～11 使用结构或类来完成会更“优雅”。编程项目 12～15 要求使用多维数组，不要求使用结构或类。要了解如何定义类和结构，请参见第 10～11 章。

1. 本项目有三个版本。

   版本 1（完全交互式）。写程序读入某个城市的月平均降雨量，再读入过去 12 个月的实际月降雨量。然后，程序打印一个良好格式化的表格，显示在过去 12 个月中，每个月的降雨量，以及它与当月平均降雨量的差值。首先给出一月平均月降雨量，再给出二月平均月降雨量，依此类推。为了获得过去 12 个月的实际降雨量，程序首先询问当前是哪一个月，再询问过去 12 个月的每月实际降雨量。输出时应正确标记月份。

   处理月份名称有多种方式。最直观的就是使用整数月份编号，并在输出时转换。在输出函数中，允许使用一个大型 switch 语句。月份输入可采用任何方式，只要用户觉得容易和舒适。

完成上述程序后，再制作它的一个增强版本。这个版本能用图表显示过去 12 个月每个月的平均降雨量和实际降雨量。图表应该与图 7.8 相似，只是每个月都应该对应两个条形图，分别标记为平均降雨量和实际降雨量。程序应询问用户喜欢用表格来输出，还是喜欢用条形图来输出，并显示他们所要求的格式。请包括一个循环，允许用户多次执行这个程序，直到他们请求终止程序。

版本 2(交互式与文件输出相结合)。为了开发一个更全面的版本，还要允许用户将表格和图表输出到一个文件。文件名由用户指定。这个版本具有与版本 1 完全相同的功能，只是增加了输出到文件的能力。读取文件名需用到 6.1 节的知识。

版本 3(用文件完成全部输入和输出)。这个版本与版本 1 相似，只是要求从文件获取输入，输出的内容也要求发送到文件。由于不涉及用户交互，所以没有循环来允许重复执行与显示。表格和图表都输出到同一个文件。如果这是课堂作业，请向老师询问文件名。

2. 十六进制数是使用 base 16 写的整数。16 个数是'0'～'9'，另外用'a'表示数字'10'，'b'表示数字'11'，'c'表示数字'12'，'d'表示数字'13'，'e'表示数字'14'，'f'表示数字'15'。例如，十六进制数 d 等同于用 base 10 写成的 13；而十六进制数 1d 等同于用 base 10 写的数字 29。请写 C++程序执行两个十六进制数的加法，每个数最多 10 位。加法结果超过 10 位就输出消息“Addition Overflow”，而不要输出加法结果。使用字符数组存储十六进制数。用循环允许用户输入新数字来重复计算，直到用户要求终止程序。

3. 视频讲解：*Solution to Programming Project 7.3*

   写 deleteRepeats 函数用一个部分填充的字符数组作为形参，从数组中删除所有重复的字母。由于部分填充的数组要求使用两个参数，所以函数实际有两个形参：一个数组形参和一个 int 类型的形参，后者给出实际使用的数组位置(元素)数目。删除一个字母后，剩余的字母依次前移，以填补空出来的位置。这会在数组末端创建空位置，而实际使用的数组位置数目会减少。由于形参是部分填充的数组，所以 int 类型的第二个形参将指出填充了多少个数组位置。第二个形参是传引用参数，删除重复字母后，该参数将改变以反映实际使用了多少个数组位置。

   例如以下代码：

```
char a[10];
a[0] = 'a';
a[1] = 'b';
a[2] = 'a';
a[3] = 'c';
int size = 4;
deleteRepeats(a, size);
```

   代码执行之后，a[0]的值是'a'，a[1]的值是'b'，a[2]的值是'c'，而 size 的值是 3(不再关心 a[3]的值，部分填充的数组不再使用该索引变量)。

   可假定部分填充的数组只包含小写字母。将函数嵌入一个恰当的测试程序中。

4. 一组数字的“标准差”(standard deviation)衡量的是数字与平均值的差异。如标准差较小，表明数字接近平均值。如标准差较大，表明数字远离平均值。对于由 N 个数字构成的一个列表(列表中的数字表示为 x_i)，标准差 S 的定义如下：

$$S = \sqrt{\frac{\sum_{i=1}^{N}(x_i - \overline{x})^2}{N}}$$

   其中的 $\overline{x}$ 是 x_1，x_2，……等 N 个数字的平均值。请定义一个函数，获取一个部分填充的数字数组作为参数，并返回该部分填充数组中的各个数字的标准差。由于部分填充的数组需要两个参数，所以函数实际有两个形参：一个数组形参和一个 int 形参，后者给出实际使用的数组位置数目。数组中的数字应该为 double 类型。将你的函数嵌入一个恰当的测试程序中。

5. 写程序将一个整数列表读入基类型为 int 的数组。要求既能从键盘读入这个数组，也能从文件读入，具体由用户决定。如选择文件输入，程序要请求一个文件名。可假定数组中的数据项少于 50 个。程序判断实际有多个数据项。最终输出的是一个双栏列表。第一栏是数组元素列表，其中包含不重复的数组元素值。第二栏是每个元素值的出现次数统计。列表第一栏按从大到小的顺序排序。

例如，假定输入如下：

```
-12 3 -12 4 1 1 -12 1 -1 1 2 3 4 2 3 -12
```

则输出如下：

```
 N   Count
 4   2
 3   3
 2   2
 1   4
-1   1
-12  4
```

6. 本章讨论了“选择排序”技术。这里要推荐另一种排序技术，称为“插入排序”。这种排序技术与选择排序相反，因为它从数组中连续选取各个元素(每次选取一个)。然后，在一个已排好序的子数组中，将选取的元素插入正确位置(首先插入子数组的尾端，然后视情况移动它的位置)。

准备排序的数组要分解成一个“已排序子数组”和一个“未排序子数组”。最开始，已排序子数组不包含任何内容。要从未排序数组中依次选取每个元素，并将其插入已排序子数组的正确位置。
请写一个函数和一个测试程序来实现选择排序。并全面测试你的程序。

示例和提示：实现时，要使用一个外层循环，它从未排序子数组中连续选取元素；还要使用一个嵌套(内部)循环，将每个元素都插入该元素在已排序子数组中的正确位置。

最开始，已排序数组为空，未排序数组包含了原始数组的全部内容：

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
8	6	10	2	16	4	18	14	12	10

选取第一个元素 a[0](值为 8)，把它放到第一个位置。内层循环在第一种情况下不做任何事情。数组和子数组看起来就像下面这样：

已排序	未排序								
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
8	6	10	2	16	4	18	14	12	10

现在，未排序子数组的第一个元素成为 a[1]，它的值是 6。在已排序子数组中，把它插入正确位置。由于顺序被打乱，所以内层循环要交换位置 0 和位置 1 的值。结果如下：

已排序		未排序							
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
6	8	10	2	16	4	18	14	10	12

注意，已排序子数组的长度增加了 1 个位置。

为未排序子数组的第一个元素(现在是 a[2])重复这一过程，查找 a[2]在已排序子数组的正确位置，保证它的已排序状态。由于 a[2]的值比已排序子数组的最大元素还要大，所以不需要和其他元素交换位置。因此，内层循环什么事情都不做。结果如下：

已排序			未排序						
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
6	8	10	2	16	4	18	14	10	12

同样，选取第一个未排序数组元素，即 a[3]。这次在内层循环中，必须对值进行几次交换，才能将 a[3]的值移动到正确位置。具体交换过程如下：

已排序				未排序					
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
6	8	10<-->2		16	4	18	14	10	12

已排序				未排序					
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
6	8<--->2		10	16	4	18	14	10	12

已排序				未排序					
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
6<--->2		8	10	16	4	18	14	10	12

将 2 放到已排序子数组之后，结果如下所示：

已排序				未排序					
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
2	6	8	10	16	4	18	14	10	12

算法继续以这种方式工作，直到未排序数组变空，而且已排序数组中包含了原始数组的全部元素。

7. 数组可存储巨大的整数，办法是一次存储一位。例如，用数组 a 存储整数 1234 时，可将 a[0]设为 1，a[1]设为 2，a[2]设为 3，a[3]设为 4。但就本项目来说，你会发现更好的做法是反向存储各个数字；换言之，将 4 放到 a[0]中，将 3 放到 a[1]中，将 2 放到 a[2]中，将 1 放到 a[3]中。

   写一个程序来读入两个正整数，它们的长度小于或等于 20 位，输出两个整数之和。程序将数字当作 char 类型的值来读入。例如，整数 1234 用 4 个字符来读入：'1'，'2'，'3'和'4'。字符读入后转换为 int 值。int 值存储到一个部分填充的数组。你可能会发现，使用通过键盘读入的数据来填充好数组之后，最好反转元素在数组中的顺序(是否反转元素在数组中的顺序由你自己决定。两种方式都是可行的，分别有自己的优点和缺点)。

   程序执行加法采用标准笔算方式。加法结果存储在一个长度为 20 的数组中，然后将结果写到屏幕。如加法运算的结果超出允许的位数(超过 20 位)，程序应报告一条消息，指出它遇到了“整数溢出”问题。要求只需更改一个全局常量，就能更改整数的最大长度。请包括一个循环，允许用户执行任意次数的加法运算，直到用户要求终止。

8. 写程序读入一行文本，输出一个列表，列举文本中出现的所有字母，并指出每个字母的出现次数。用句点符号终止一行文本，句点作为哨兵值使用。字母要按以下顺序列出：最常用的字母、下一个最常用的字母……等等。使用两个数组，一个容纳整数，另一个容纳字母。可假定输入的全是小写字母。例如以下输入：

   ```
   do be do bo.
   ```

   程序生成的输出如下：

   ```
   Letter Number of Occurrences
      o          3
      d          2
      b          2
      e          1
   ```

   程序需要根据整数数组中的值来进行排序。这要求你修改图 7.12 给出的 sort 函数。不修改函数，就不能用 sort 来解决这个问题。如果这是课堂作业，请询问老师输入/输出是用键盘和屏幕完成，还是用文件完成。如果用文件，问老师文件名是什么。

9. 写程序将一手共计 5 张扑克牌分类为以下任何一个类别：nothing(杂牌)、one pair(一对)、two pairs(两

对)、three of a kind(三张，即 3 张牌的点数相同)、straight(顺子，即 5 张牌点数连续)、flush(同花，即 5 张牌同一种花色，比如都是黑桃)、full house(一对加三张)、four of a kind(四张，即 4 张牌的点数相同)、straight flush(同花顺，既是同花，又是顺子)。使用两个数组，一个容纳牌点(比如 2，3，……，J，Q，K，A 等)，另一个容纳花色。请包括一个循环，允许用户对下一手牌进行分类，直到要求终止。

10. 写程序让两个用户玩 Tic-Tac-Toe(三连棋)游戏。程序允许玩家 X 和玩家 O 交替移动棋子。

    程序显示下面这样的棋盘：

```
1 2 3
4 5 6
7 8 9
```

    玩家为了下一步棋，必须输入希望标记的位置编号。每下一步棋，程序都要显示发生变化的棋盘。例如，下到中途的一盘棋可能是：

```
X X O
4 5 6
O 8 9
```

11. 写程序为航班分配乘客座位。假定一架小飞机的座位编号如下：

```
1  A B  C D
2  A B  C D
3  A B  C D
4  A B  C D
5  A B  C D
6  A B  C D
7  A B  C D
```

    程序应该显示可用座位的分布图，用'X'来标记一个已分配的座位。例如，在座位 1A，2B 和 4C 被订出去之后，应显示以下座位分布图：

```
1  X B  C D
2  A X  C D
3  A B  C D
4  A B  X D
5  A B  C D
6  A B  C D
7  A B  C D
```

    在显示可用座位的分布图后，程序提示用户输入希望的座位。当用户输入自己选中的座位后，程序对可用座位分布图进行更新。程序应该一直执行下去，直到所有座位都被预订，或者用户表示程序应该终止。如果用户指定的一个座位已被预订，程序指出该座位不可用，要求重新选择。

12. 写程序像图 7.8 那样接收输入并输出一个条形图。但程序应输出垂直条形图(称为“柱形图”)。一个二维数组也许非常有用。

13. 英国数学家 John Horton Conway 发明了“生命游戏”(Game of Life)。虽然不是传统意义上的“游戏”，但只需指定少数几条规则，它就能产生非常有趣的行为。这个项目要求写一个程序，允许用户指定一个初始配置。程序将遵循生命规则(稍后会简单地列出)，显示初始配置连续变化的行为。

    LIFE 是生活在离散二维世界中的生命体。虽然这个世界实际是无限的，但我们无法享受这种奢侈，所以用数组模拟 80 字符宽和 22 字符高的二维世界。如使用的屏幕较大，不妨修改这个尺寸，总之以满屏显示为宜。

    在数组中，每个存储位置都能容纳一个 LIFE 元胞。世代(generation)用于标记时间的流逝。每个世代都为 LIFE 社会带来了生与死。生死规则如下。

    - 定义每个元胞都有 8 个邻居(neighbor)元胞。这些邻居在它的正上方、正下方、右侧、左侧、左上方、右上方、左下方以及右下方。
    - 如果一个已被占据的元胞具有零个或者一个邻居，会因为孤独(loneliness)而死亡。如果一个已被占据的元胞具有 3 个以上的邻居，会因为拥挤(overcrowding)而死亡。
    - 如果一个空元胞恰好有 3 个被占据的邻居元胞，就诞生一个新元胞，它会取代那个空元胞。
    - 生与死在世代交替时瞬时发生。无论因为什么原因而濒死的一个元胞可能有利于其他元胞的

诞生。但一个新诞生的元胞不能让一个濒死元胞复苏，一个元胞的死亡也不能阻止另一个元胞的死亡(即使本地生命体数量减少了)。

注意：有的初始配置的规模相当小，有的则占据了一大片区域。建议为了优化文本输出，使用一个 80 列和 22 行的矩形 char 数组来存储 LIFE 世界的不断发展的世代。使用星号表示一个存活的元胞，使用空格表示一个空的(或者死亡的)元胞。如屏幕支持更大的显示区域，请尽量使用整个屏幕。

示例：

```
***
```

变成：

```
*
*
*
```

再变成：

```
***
```

如此反复。

建议：查找稳定配置。也就是说，查找进化模式不断重复的生命群体。每次重复所经历的世代数称为“周期”(period)。有些配置固定不变，它们永远不会发生改变。可考虑做的一个项目是查找这种配置。

提示：定义名为 generation 的 void 函数，它获取我们称为 world 的一个数组，这是一个 80 列和 22 行的 char 数组。该数组已包含了初始配置。函数对数组进行扫描，并修改元胞，根据前面列出的规则来标记元胞的生与死。这要求依次检查每个元胞，要么杀死元胞，要么保持存活。如果是一个空元胞，就判断是否应该诞生一个新元胞。应该设计一个 display 函数，它接收数组 world 作为参数，并在屏幕上显示数组。每次显示一个世代时，最好留一些延迟时间。你可考虑只有在按了回车键之后，才让程序生成和显示下一代。虽然也可以让程序自动连续输出，但对于这个程序来说，自动化是不必要的。

14. 重做(或第一次做)第 6 章的编程项目 10。程序首先将所有男孩和女孩名字从文件加载到单独的数组中。然后，在数组中查找目标名字，而不是直接从文件中查找。

15. 重做(或第一次做)第 6 章的编程项目 11。程序不硬性规定只能描绘 4 个整数的柱形图，而是应该支持对最多 100 个整数的一个数组进行描绘。柱形图在水平和垂直方向上都要按比例缩放，最终的图像要占据一个 400×400 像素的区域。可限制数组中的所有整数都是非负的。使用哨兵值-1 来标识要描绘的整数值的结束。例如，为了创建一幅柱形图来描绘值 20，40，60 和 120，你的程序应该对下面这个数组进行处理：

```
a[0] = 20
a[1] = 40
a[2] = 60
a[3] = 120
a[4] = -1
```

用不同的值(最多可以有 100 个值)来创建几幅柱形图，查看最终的 SVG 文件，确定它们能正确描绘。

16. “记忆匹配”(memory matching game)是小孩子喜欢玩的一个益智游戏。首先准备好一墩牌，它由几个“对子”组成。例如，假定一墩牌有 6 张牌，2 张是“1”，2 张是“2”，另外 2 张是“3”。现在洗好这墩牌，然后全部反扣在桌面上。玩家选择 2 张反扣着的牌，把它们翻过来，如果恰好是“对子”，就保持它们现在的样子，不要再反扣回去。相反，如果不是“对子”，就把它们反扣回去。一直这样翻牌，直到最后所有牌都正面朝上。

写程序来玩这个记忆匹配游戏。要求使用 16 张牌，它们按 4×4 排列，用 1～8 的数字成对标记这些牌。程序应该允许玩家通过一个坐标系统来指定要翻的牌。

例如，假定现在的牌面布局如下：

```
       1 2 3 4
---------------
```

```
1 | 8 * * *
2 | * * * *
3 | * 8 * *
4 | * * * *
```

所有牌都反扣着，除了“8”这对牌。“8”这对牌位于坐标 (1,1) 和 (2,3) 处。为了隐藏被临时翻起来的牌，请输出大量换行符，强迫旧图离开屏幕。

提示：用一个二维数组描述牌的布局，用另一个二维数组描述一张牌是面朝上还是朝下。写一个函数来“洗牌”，它反复随机选择两张牌，并交换它们的位置。随机数生成已在第 4 章讨论。

17. 游泳学校有两名游泳教练，Jeff 和 Anna。他们的当前的日程表如下。每个“X”都代表那个时间段已经排了 1 小时的课，没有“X”则表示教练在那个时间段空闲。

Jeff	Monday	Tuesday	Wednesday	Thursday
11-12	X	X		
12-1		X	X	X
1-2		X	X	
2-3	X	X	X	

Anna	Monday	Tuesday	Wednesday	Thursday
11-12	X	X		X
12-1		X		X
1-2	X	X		
2-3	X		X	X

写程序用数组存储上述日程表。创建一个主菜单，允许用户将两个教练的某个时间段标记为忙或空闲。另外，添加一个选项在屏幕上输出最新的日程表。接着，添加一个选项来输出上“一对一课程”(individual lesson)可用的所有时间段(至少有一个教练空闲的时间段)。最后，添加一个选项来输出上“集体课”(group lesson)可用的所有时间段(两个教练都空闲的时间段)。

18. 修改编程项目 17，添加在文件中加载和保存日程表的菜单选项。

19. 传统密码输入方案容易遭受“肩头偷窥”，攻击者从用户背后偷看他输入的密码或 PIN 码，记下来以便稍后访问他的账号。对抗这种攻击的一个办法是使用随机“质询-应答”系统。在这种系统中，用户每次都根据一个秘密来输入不同的信息，从而响应系统生成的“质询”。例如在以下方案中，用户输入密码是 5 位 PIN 码(00000~99999)。每一位都对应随机数 1，2 或者 3。用户输入和自己的 PIN 码对应的随机数，而不是输入真实的 PIN 码。例如，假定真实的 PIN 码是 12345。为了验证用户身份，屏幕上会显示以下“质询”：

```
PIN: 0 1 2 3 4 5 6 7 8 9

NUM: 3 2 3 1 1 3 2 2 1 3
```

用户要输入 23113 而不是 12345。即使攻击者拦截到了输入，密码也不会泄漏，因为 23113 还可对应于其他 PIN 码，比如 69440 或 70439。用户下次登录时，会生成一组不同的随机数，比如：

```
PIN: 0 1 2 3 4 5 6 7 8 9

NUM: 1 1 2 3 1 2 2 3 3 3
```

程序应模拟身份验证过程。在程序中存储一个真实的 PIN 码。程序用一个数组将随机数分配给 0 到 9 的数位。在屏幕上输出随机数，从用户处获取输入，最后报告用户的应答是否和 PIN 码匹配。

20. 美国社会安全局维护着一份真实的人寿表，其中包括每个美国人可能死亡的时间(请访问 http://www.ssa.gov/OACT/STATS/table4c6.html)。本书配套资源有一个 LifeDeathProbability.txt 文件，其中存储了 2009 年美国人的死亡率。每行三个值，包括年龄、男性死亡率和女性死亡率。例如，文件前 5 行如下所示：

```
0 0.006990 0.005728
1 0.000447 0.000373
2 0.000301 0.000241
3 0.000233 0.000186
4 0.000177 0.000150
```

这表明 3 岁女童死亡率是 0.000186。

写程序将数据读入数组(可能需要多个数组)。让用户输入性别和年龄。程序根据数据模拟用户还能活多久。生成 0~1 的随机数，小于等于死亡率就预测只能活到当前年龄。随机数大于死亡率，就将年龄增大 1 岁，生成新随机数重复计算。模拟到 120 岁就停止计算，并预测用户能活到 120 岁。程序只是模拟，如果每次运行都更改随机数生成器的种子值，那么每次都会有不同的结果。

第 8 章　字符串和向量

普娄尼阿斯："你读什么呢？殿下。"

哈姆雷特："字，字，字。"

——*威廉·莎士比亚(1564—1616)，《哈姆雷特》*

概述

本章讨论使用数组或关于数组的两个主题：字符串和向量。虽然字符串和向量密切相关，但这个关系并非总是那么明显，而且两个主题互不依赖。可以先学字符串，再学向量。也可以先学向量，再学字符串。

8.1 节和 8.2 节分别介绍了两种字符串类型，它们的值用于表示"Hello"这样的字符串。8.1 节讨论的其实是基类型为 char 的数组，它将字符串存储到数组中，用空字符'\0'标记字符串结束。这是表示字符串的一种古老方式，是 C++从 C 语言继承的，称为 C 字符串。虽然很古老，但起码要对它有一个基本认识，否则在 C++中很难进行任何形式的字符串处理。例如，引号字符串(比如"Hello")在 C++中就作为 C 字符串实现。

ANSI/ISO C++标准支持更现代的字符串处理机制，也就是使用 string 类，这是本章讨论的第二种字符串类型(8.2 节)。

向量可被视为程序运行时长度可变的数组。在 C++中，一旦程序创建好数组，该数组的长度就不能改变。向量作用和数组一样，只是长度能在运行时改变。

预备知识

8.1 节/8.2 节(主要讨论字符串)和 8.3 节(主要讨论向量)相互独立。可选择先学向量，再学字符串。

讲解 C 字符串的 8.1 节基于第 2 章～第 6 章、第 7 章的 7.1 节、7.2 节和 7.3 节的知识。

讲解 string 类的 8.2 节基于 8.1 节、第 2 章～第 6 章以及第 7 章的 7.1 节、7.2 节和 7.3 节的知识。

讲解向量的 8.3 节基于第 2 章～第 6 章以及第 7 章的 7.1 节、7.2 节和 7.3 节的知识。

8.1　字符串的数组类型

任何事情，都必须考虑后果[①]。

——*让·德·拉封丹[②](1621—1695)，《拉封丹寓言》第三卷 (1668)*

本节介绍表示字符串的一种方式，C++从 C 语言继承而来。8.2 节将介绍 string 类，那是表示字符串的一种更现代的方式。虽然本节描述的字符串类型较"古老"，但仍在广

① 这行诗中，后果原文是 end，隐喻用于结束字符串的'\0'。——译注

② 让·德·拉·封丹(Jean de la Fontaine)，法国古典文学的代表作家之一，著名的寓言诗人。他的作品经后人整理为《拉·封丹寓言》，与古希腊著名寓言诗人伊索的《伊索寓言》及俄国著名作家克雷洛夫所著的《克雷洛夫寓言》并称为世界三大寓言。主要著作有《寓言诗》《故事诗》《普叙赫和库比德的爱情》等。他被 19 世纪法国著名文学评论家泰纳誉为"法国的荷马"。雨果的《巴黎圣母院》和莫泊桑的《一生》都提到他是法国古典文学作家中著名的诗人。——编注

泛使用，而且是 C++语言密不可分的一部分。

C 字符串值和 C 字符串变量

为了表示字符串，一个办法是表示成基类型为 char 的数组。例如，字符串"Hello"很容易表示成含有 6 个索引变量的字符数组。其中，5 个索引变量表示"Hello"的 5 个字母，另一个索引变量表示字符'\0'，即结束标记。字符'\0'称为**空字符**，作为结束标记使用，因其有别于任何“真正”的字符。结束标记允许程序每次从数组读取一个字符。一旦遇到结束标记'\0'，就知道应该停止读取。以这种方式存储的字符串(以'\0'终止的字符数组)称为“C 字符串”。

在程序中写'\0'时，虽然要使用两个符号，但和换行符'\n'一样，字符'\0'实际是一个字符值。和其他任何字符值一样，'\0'可用 char 类型的变量来存储，或者用字符数组的索引变量来存储。

> **空字符'\0'**
>
> 空字符'\0'标记字符数组中存储的 C 字符串的结束。如果字符数组以这种方式使用，该数组就称为 C 字符串变量。虽然空字符'\0'要写成两个符号，但实际只代表一个字符，可以用 char 类型的变量或者字符数组的索引变量来存储它。

其实前面已经用过 C 字符串。在 C++中，作为字面值使用的字符串(比如"Hello")是以 C 字符串的形式存储的，只是这个细节很少有人注意到而已。

C 字符串变量本质上是字符数组。因此，以下数组声明将生成一个 C 字符串变量，它最多能存储由 9 个(或更少)字符构成的 C 字符串值。

```
char s[10];
```

虽然数组长度是 10，但只能有 9 个字母，另外有一个空字符'\0'标记字符串结束。

C 字符串变量是部分填充的字符数组。和其他部分填充的数组一样，C 字符串变量使用从索引变量 0 开始的索引变量。然而，C 字符串变量不用一个 int 变量来跟踪数组中已经使用了多少个存储位置。相反，字符串变量在 C 字符串的最后一个字符之后添加特殊符号'\0'。所以，如果 s 包含字符串"Hi Mom!"，那么数组元素的填充情况如下：

s[0]	s[1]	s[2]	s[3]	s[4]	s[5]	s[6]	s[7]	s[8]	s[9]
H	i		M	o	m	!	\0	?	?

字符'\0'作为哨兵值标记 C 字符串结束。读取 C 字符串中的字符时，如果首先读取索引变量 s[0]，再读取 s[1]，再读取 s[2]，依此类推。一旦遇到符号'\0'，就表明抵达了 C 字符串的末尾。由于符号'\0'总是占用一个数组元素，所以数组能容纳的最大字符串长度是数组长度减 1。

C 字符串变量与普通字符数组的区别在于，C 字符串变量必须在 C 字符串值的尾部包含空字符'\0'。但这只是数组的两种不同的用法，数组的本质并没有改变。C 字符串变量本质上还是字符数组，只是以不同的方式使用。

可在声明 C 字符串变量时初始化它，如下例所示：

```
char myMessage[20] = "Hi there.";
```

注意，将 C 字符串赋给 C 字符串变量时，不需要将整个数组都填满。

C 字符串变量声明

C 字符串变量本质上是字符数组，只是用法不同。C 字符串变量要以声明数组的标准方式声明成一个字符数组。

语法

```
char ArrayName[Maximum_C_string_Size + 1];
```

示例

```
char myCString[11];
```

之所以要加 1，是考虑到还有空字符'\0'，它终止存储在数组中的任何 C 字符串。例如，上例的 C 字符串变量 myCString 最多容纳由 10 个(或更少)字符构成的一个 C 字符串。

初始化 C 字符串变量时可省略数组长度。C++会自动使 C 字符串变量的长度等于引号字符串的长度加 1(多出来的索引变量用于存储'\0')。例如：

```
char shortString[] = "abc";
```

等价于：

```
char shortString[4] = "abc";
```

一定不要将以下初始化语句：

```
char shortString[] = "abc";
```

与以下初始化语句混淆：

```
char shortString[] ={'a', 'b', 'c'};
```

两者不等价。第一个会在字符'a'，'b'和'c'之后添加空字符'\0'，第二个不会。

初始化 C 字符串变量

C 字符串变量可在声明时初始化，如下例所示：

```
char yourString[11] = "Do Be Do";
```

以这种方式初始化，会在指定 C 字符串末尾自动添加空字符'\0'。

如果省略方括号中的数字，C 字符串变量的长度就会自动设置成 C 字符串的长度加 1。例如，以下语句会将 myString 声明为具有 9 个索引变量(C 字符串"Do Be Do"的 8 个字符加空字符'\0')：

```
char myString[] = "Do Be Do";
```

C 字符串变量其实就是数组，所以也有索引变量，可像其他任何数组的索引变量那样使用。例如，假定程序包含以下 C 字符串变量声明：

```
char ourString[5] = "Hi";
```

那么程序中将包含以下索引变量：ourString[0]，ourString[1]，ourString[2]，

ourString[3]和 ourString[4]。例如，以下代码将 ourString 中的 C 字符串值更改为一个长度相同，但全由'X'字符构成的 C 字符串：

```
int index = 0;
while (ourString[index] != '\0')
{
    ourString[index] = 'X';
    index++;
}
```

填充这些索引变量时要仔细，千万不要将空字符'\0'替换成其他值。假如数组中丢失了'\0'值，它的行为就不再是一个 C 字符串变量。例如，以下语句会更改数组 happyString，使其不再包含一个 C 字符串：

```
char happyString[7] = "DoBeDo";
happyString[6] = 'Z';
```

执行上述代码后，数组 happyString 仍然包含 C 字符串"DoBeDo"的 6 个字母。但 happyString 不再包含标记 C 字符串结束的空字符'\0'。许多字符串处理函数都完全依赖'\0'标记 C 字符串值的结束。

作为另一个例子，请思考前面对 C 字符串变量 ourString 中的字符进行更改的 while 循环。循环会连续更改字符，直到遇到'\0'。假如循环永远遇不到'\0'，就会将大量内存位置更改为不希望的值，而这可能导致程序异常。为保险起见，最好像下面这样重写 while 循环。它确保即使丢失空字符'\0'，循环也不会更改超出数组有效范围的内存位置：

```
int index = 0;
while ( (ourString[index] != '\0') && (index < SIZE) )
{
    ourString[index] = 'X';
    index++;
}
```

SIZE 是已定义常量，等于 ourString 数组的声明长度。

陷阱：为 C 字符串使用=和==

C 字符串值和 C 字符串变量有别于其他数据类型的值和变量，许多常规操作都不适合 C 字符串。不能用=对 C 字符串变量进行赋值。用==测试 C 字符串相等性也得不到你希望的结果。原因很简单，因为 C 字符串和 C 字符串变量本质上是数组。

为 C 字符串变量赋值不像为其他类型的变量赋值那么简单。以下语句非法：

```
char aString[10];
aString = "Hello";      ←—————— 非法！
```

虽然可在声明时用等号为 C 字符串变量赋值，但在程序的其他任何地方都不能这样做。从技术上说，在声明时使用等号，就像下面这样：

```
char happyString[7] = "DoBeDo";
```

它其实是一个初始化操作，而不是赋值操作。要为 C 字符串变量赋值，必须采取其他方式。

可采取许多不同的方式为 C 字符串变量赋值。最简单的是使用预定义函数 strcpy，如下所示：

```
strcpy(aString, "Hello");
```

这会将 aString 的值设为"Hello"。遗憾的是，strcpy 函数的这个版本不检查复制的字符串是否超过字符串变量(第一个参数)的长度。

许多(但并非全部)C++版本还提供了 strcpy 的一个更安全的版本。这个安全版本要拼写成 strncpy(多了一个 n)。strncpy 函数要获取第 3 个参数，指定最多能复制多少字符。例如下面的代码：

```
char anotherString[10];
strncpy(anotherString, aStringVariable, 9);
```

使用这个 strncpy 函数，最多能从 C 字符串变量 aStringVariable 中复制 9 个字符(包括'\0')，无论 aStringVariable 中的字符串有多长。

也不能在表达式中使用操作符==测试两个 C 字符串是否相等。更糟的是，可以为 C 字符串使用==，但作用不是测试 C 字符串是否相等。所以，用==测试两个 C 字符串的相等性可能得到错误结果，而且编译器不报告任何错误！要测试两个 C 字符串是否相等，可以使用预定义函数 strcmp。例如下面的代码：

```
if (strcmp(cString1, cString2))
   cout << "The strings are NOT the same.";
else
   cout << "The strings are the same.";
```

注意，strcmp 函数的工作方式可能和你想的不同。两个字符串不匹配，结果反而是 true。strcmp 函数每次比较 C 字符串参数的一个字符。任何时候只要 cString1 的一个字符的数值编码小于 cString2 的对应字符的数值编码，测试就会停止，并返回一个负数。如果 cString1 的字符大于 cString2 的对应字符，则返回一个正数(strcmp 的有些实现返回字符编码的差值，但不应依赖于此)。两个 C 字符串完全相同，则返回 0。字符比较依据的是**词典顺序(lexicographic order)**。它的重点是，两个字符串采用全部大写或小写的形式，词典顺序就和字母顺序一样。

基于词典顺序，在 C 字符串比较结果是小于、大于或者等于的情况下，strcmp 分别返回负值、正值或零。在 if 或循环语句中将 strcmp 作为布尔表达式使用，在字符串不相等的前提下，返回的所有非零值都被转换成 true；在字符串相等的前提下，返回的零值被转换成 false。所以在测试 C 字符串的相等性时，务必记住这一反转逻辑。符合标准的 C++编译器提供了一个更安全的 strcmp 版本，它的第 3 个参数指定了要比较的最大字符数。

strcpy 和 strcmp 在头文件为<cstring>的库中，要使用这些函数，必须在接近文件顶部的位置插入以下代码：

```
#include <cstring>
```

strcpy 和 strcmp 不需要以下或其他类似语句(尽管程序其他部分可能需要)：[①]

```
using namespace std;
```

■

① 第 12 章会讲到，strcpy 和 strcmp(以及<cstring>中的其他字符串函数)的定义放在全局命名空间，而非放在 std 命名空间，所以不需要 using 指令。

<cstring>库

声明和初始化 C 字符串不需要任何 include 或 using 预编译指令。但在处理 C 字符串时，不可避免地要用到来自<cstring>库的某些预定义字符串函数。所以，使用 C 字符串时，通常要在接近源代码文件开头的地方包括以下预编译指令：

```
#include <cstring>
```

<cstring>中的其他函数

图 8.1 总结了头文件为<cstring>的库所提供的常用函数。使用这些函数需在接近文件顶部的位置插入以下代码：

```
#include <cstring>
```

与函数 strcpy 和 strcmp 相似，<cstring>中的其他所有函数也不需要以下语句或其他类似语句(虽然程序其他部分可能需要)：

```
using namespace std;
```

上一节已讨论了 strcpy 和 strcmp。strlen 函数很容易理解和使用。例如，strlen("dobedo")返回 6，因为"dobedo"有 6 个字符。

strcat 函数用于连接两个 C 字符串；换言之，它首尾连接两个较短的 C 字符串来构成一个较长的。第一个参数必须是 C 字符串变量。第二个是最终能求值为 C 字符串值的任何东西，比如一个引号字符串。结果放到第一个参数指定的 C 字符串变量中。例如以下代码：

```
char stringVar[20] = "The rain";
strcat(stringVar, "in Spain");
```

它会将 stringVar 的值变成"The rainin Spain"。如同这个例子所演示的，在连接 C 字符串时，注意预留空格(rain 和 in 之间应该有一个空格，但本例忘记预留一个)。

C 字符串实参和形参

C 字符串变量是数组，所以函数的 C 字符串形参实际是数组形参。

和任何数组形参一样，函数要更改 C 字符串形参的值，最保险的做法是包括一个附加的 int 形参，给出 C 字符串变量的声明长度。

另一方面，如果函数只是使用 C 字符串实参所包含的值，而不更改那个值，就没必要包括另一个形参来给出 C 字符串变量的声明长度或者 C 字符串变量数组的实际填充量。用空字符'\0'检测 C 字符串变量中存储的 C 字符串值的结束。

如图 8.1 所示，许多(但并非全部)C++版本都提供了 strcpy，strcat 和 strcmp 函数的更安全的、总共三个参数的版本。另外要注意，这三个参数的版本在拼写时都添加了字母 n：strncpy，strncat 以及 strncmp。

图 8.1　<cstring>的部分预定义 C 字符串函数

函数	说明	提示
strcpy(*Target_StringVar*, *SrcString*)	将 C 字符串值 *SrcString* 复制到 C 字符串变量 *Target_StringVar* 中	不检查 *Target_StringVar* 是否有足够大的空间来容纳 *SrcString* 值
strncpy(*Target_StringVar*, *SrcString*, *Limit*)	与两个参数的 strcpy 相同，只是最多复制 *Limit* 个字符	如果慎重选择 *Limit*，它比两个参数的 strcpy 版本更安全。并非所有 C++版本都实现
strcat(*Target_StringVar*, *SrcString*)	将 C 字符串值 *SrcString* 连接到 C 字符串变量 *Target_StringVar* 中的 C 字符串的末尾	不检查 *Target_StringVar* 是否有足够大的空间来容纳连接结果
strncat(*Target_StringVar*, *SrcString*, *Limit*)	与两个参数的 strcat 相同，只是最多附加 *Limit* 个字符	如果慎重选择 *Limit*，它比两个参数的 strcat 版本更安全。并非所有 C++版本都实现
strlen(*SrcString*)	返回与 *SrcString* 的长度相等的整数(空字符'\0'不计算在内)	
strcmp(*String_1*, *String_2*)	如果 *String_1* 和 *String_2* 内容相同，就返回 0。如果 *String_1* 小于 *String_2*，就返回小于 0 的值。如果 *String_1* 大于 *String_2*，就返回大于 0 的值。总之，*String_1* 和 *String_2* 内容不同，就返回非零值。依据词典顺序	如果 *String_1* 等于 *String_2*，函数返回 0，它会转换成 false。注意这可能刚好和你想象的相反
strncmp(*String_1*, *String_2*, *Limit*)	与两个参数的 strcmp 相同，只是最多比较 *Limit* 个字符	如果慎重选择 *Limit*，它比两个参数的 strcmp 版本更安全。并非所有 C++版本都实现

陷阱：使用 strcpy 复制 C 字符串越界

视频讲解：*Dangers of Strcpy*

C 和 C++编程的一个常见错误是用 strcpy 函数将较大 C 字符串复制到较小 C 字符串。这很危险，因为 strcpy 对能复制的数据量不设限。它将一切从来源字符串复制到目标字符串，直至遇到空字符。如来源字符串比目标字符串大，复制就会越界。下面是一个例子：

```
void copyString(char source[])
{
   char target[5];
   strcpy(target, source);
```

```
    // 在这里使用 target 字符串
}
```

很简单，如 C 字符串 source 超过 5 个字符，数据就会复制到 target 数组后面的内存位置，发生程序崩溃或其他莫名其妙的事情。恶意用户甚至能利用这个漏洞攻击你的系统。由于问题过于严重，以至于某些编译器拒绝编译使用了 strcpy 的代码，除非你不顾警告。假定编译器允许使用 strcpy，修正问题的一个方案是只复制小于 5 个字符的 C 字符串。下例试图防止超出 C 字符串的长度限制：

```
void copyString(char source[])
{
    char target[5];
    signed char length; // 取值范围-128 到+127
    length = strlen(source);
    if (length < 5)
        strcpy(target, source);
}
```

这个版本用一个 signed char 来存储 C 字符串长度。由于只是创建长度为 5 的数组，而 signed char 能存储最大为+127 的值，所以这似乎是一个合理的方案。这个版本确实能正确处理小的来源字符串，但输入一个长度为 145 的来源字符串会发生什么？strlen 会返回 145，这超出了 signed char 的限制，造成溢出，拷贝到 length 中的将是一个负值，进而造成 if 测试通过，将来源数据错误地复制到目标数组。为彻底避免这些问题，应将 length 定义为 int(这样在变量中存储的值肯定和 strlen 返回的相同)，再用 strncpy 限制最大复制长度。还有一个方案就是使用 8.2 节讨论的 string 类。■

√ 自测题

1. 以下哪些声明是等价的？

```
char stringVar[10] = "Hello";
char stringVar[10] = {'H', 'e', 'l', 'l', 'o', '\0'};
char stringVar[10] = {'H', 'e', 'l', 'l', 'o'};
char stringVar[6] = "Hello";
char stringVar[] = "Hello";
```

2. 运行以下代码后，singingString 中将存储什么 C 字符串？

```
char singingString[20] = "DoBeDo";
strcat(singingString, " to you");
```

   假定代码嵌入一个完整和正确的程序，而且已 include 了<cstring>。

3. 以下代码错在哪里？

```
char stringVar[] = "Hello";
strcat(stringVar, " and Good-bye.");
cout << stringVar;
```

   假定代码嵌入一个完整和正确的程序，而且已 include 了<cstring>。

4. 假定 strlen 函数(返回字符串参数的长度)尚未定义。请给出 strlen 的函数定义。注意，strlen 只有一个 C 字符串参数。不要添加额外的参数，它们用不着。

5. 对于用以下语句声明的字符串变量，它能容纳的最大字符串长度是多少？请解释原因。

```
char s[6];
```

6. 以下每个字符和字符串常量中有多少个字符？

a. `'\n'`
b. `'n'`
c. `"Mary"`
d. `"M"`
e. `"Mary\n"`

7. 既然字符串不过是 char 数组，本书为什么还要提出警告，称不能混淆以下两个声明及初始化语句？

```
char shortString[] = "abc";
char shortString[] = {'a', 'b', 'c'};
```

8. 给定以下字符串变量声明和初始化语句，写一个循环将'X'赋给这个字符串变量的所有存储位置，要求长度保持不变。

```
char ourString[15] = "Hi there!";
```

9. 给定以下 C 字符串变量声明，SIZE 是已定义的常量：

```
char ourString[SIZE];
```

假定 ourString 已在程序的其他地方赋值。以下循环将 ourString 的所有存储位置重新赋值为'X'，同时保持长度不变：

```
int index = 0;
while (ourString[index] != '\0')
{
    ourString[index] = 'X';
    index++;
}
```

回答以下两个问题。

a. 解释上述代码为什么有可能篡改不属于该数组的内存。
b. 修改这个循环，防止因为不慎而篡改不属于该数组的内存。

10. 写代码来使用一个库函数，将字符串常量"Hello"复制到下面声明的字符串变量。务必使用#include 来包含必要的头文件，以获得你所使用的函数的声明。

```
char aString[10];
```

11. 运行以下代码会得到什么输出结果？(和前面一样，假定嵌入一个完整而正确的程序)

```
char song[10] = "I did it ";
char franksSong[20];
strcpy ( franksSong, song );
strcat ( franksSong, "my way!");
cout << franksSong << endl;
```

12. 以下代码有什么问题？

```
char aString[20] = "How are you? ";
strcat(aString, "Good, I hope.");
```

C 字符串输入和输出

C 字符串可用插入操作符<<来输出。事实上，我们已对引号字符串进行了这样的操作。可采用相同的方式使用 C 字符串变量。例如：

```
cout << news << " Wow.\n";
```

其中，news 是 C 字符串变量。

还可用输入操作符>>填充 C 字符串变量，但要注意一点：和其他所有类型的数据一样，以这种方式读取 C 字符串时，所有空白字符(空格、制表符和换行符)都会被忽略。除此之外，每次读取输入，都会在下一个空格或换行符处停止。例如以下代码：

```
char a[80], b[80];
```

```
cout << "Enter some input:\n";
cin >> a >> b;
cout << a << b << "END OF OUTPUT\n";
```

嵌入一个完整的程序后，上述代码会生成如下所示的会话：

```
Enter some input:
Do be do to you!
DobeEND OF OUTPUT
```

也就是说，C 字符串变量 a 和 b 分别只接收输入的一个单词：a 接收 C 字符串值"Do"，因为 Do 之后的输入字符是一个空格；b 接收"be"，因为 be 之后也是空格。

如希望程序读取整行输入，可用提取操作符>>每次读入一个单词。但这会十分繁琐，而且仍然不会读入行中的空格。有一个简单办法可读入整行输入并将最终的 C 字符串放到一个 C 字符串变量中，这就是使用预定义成员函数 getline，所有输入流(比如 cin 或文件输入流)都有该成员函数。getline 函数有两个参数。第一个接收输入的 C 字符串变量，第二个是整数，一般是 C 字符串变量的声明长度。第二个参数指定 getline 最多能填充 C 字符串变量中的多少个数组元素。例如以下代码：

```
char a[80];
cout << "Enter some input:\n";
cin.getline(a, 80);
cout << a << "END OF OUTPUT\n";
```

嵌入一个完整程序后，上述代码会生成如下所示的对话：

```
Enter some input:
Do be do to you!
Do be do to you!END OF OUTPUT
```

使用 cin.getline 函数，整行都会读入。遇到行尾即停止读入，即使最终的 C 字符串短于由第二个参数指定的最大字符数。

执行 getline 时，一旦在 C 字符串数组中填充了由第二个参数指定的字符数，就会停止读入，即使当时尚未遇到行尾。例如以下代码：

```
char shortString[5];
cout << "Enter some input:\n";
cin.getline(shortString, 5);
cout << shortString << "END OF OUTPUT\n";
```

将其嵌入一个完整的程序，上述代码会生成如下所示的对话：

```
Enter some input:
dobedowap
dobeEND OF OUTPUT
```

注意，会在 C 字符串变量 shortString 中读入 4 个(而不是 5 个)字符，即使第二个参数是 5。这是由于空字符'\0'占一个数组位置。每个 C 字符串在存储到 C 字符串变量中时，都会以空字符结尾，该字符肯定占一个数组位置。

前面讲述的是 cout 和 cin 的 C 字符串输入和输出技术，这些技术同样适合文件输入和输出。输入流 cin 可替换成连接到文件的输入流，输出流 cout 可替换成连接到文件的输出流(文件 I/O 已在第 6 章讲述)。

getline

成员函数 getline 读取整行输入，将由那一行的字符构成的 C 字符串放到 C 字符串变量中。

语法

```
Input_Stream.getline(stringVar, Max_Characters + 1);
```

从 *Input_Stream* 流读取单行输入，生成的 C 字符串放到字符串变量 *stringVar* 中。如该行长度超过 *Max_Characters*，只会读入该行的前 *Max_Characters* 个字符(之所以加 1，是因为每个 C 字符串都以末尾的空字符'\0'结束，所以 *stringVar* 存储的字符串的长度要比读入的字符数多 1)。

示例

```
char oneLine[80];
cin.getline(oneLine, 80);
```

(可用连接到文本文件的输入流代替 cin。)

√ 自测题

13. 给定以下代码(假定嵌入一个完整而正确的程序后再运行)：

```
char a[80], b[80];
cout << "Enter some input:\n";
cin >> a >> b;
cout << a << '-' << b << "END OF OUTPUT\n";
```

如果对话像下面这样开始，下一行输出是什么？

```
Enter some input:
The
   time is now.
```

14. 给定以下代码(假定嵌入一个完整而正确的程序后再运行)：

```
char myString[80];
cout << "Enter a line of input:\n";
cin.getline(myString, 6);
cout << myString << "<END OF OUTPUT";
```

如果对话像下面这样开始，下一行输出是什么？

```
Enter a line of input:
May the hair on your toes grow long and curly.
```

C 字符串到数值的转换和可靠输入

C 字符串"1234"有别于数字 1234。前者是字符序列，后者是数字。我们在日常生活中以相同方式书写它们，不关心这一区别。但在 C++程序中却不能忽视这一区别。要做算术，需要的是 1234，而不是"1234"。要为数字 1234 添加千分位逗号，需要将 C 字符串"1234"转换成 C 字符串"1,234"。设计数值输入时，通常需要将输入作为字符串来读取，编辑字符串，再将字符串转换成数字。例如，如果希望程序读取一个货币金额，输入也许会、也许不会以美元符号开始。如果程序要读取百分比，输入也许会(也许不会)在末尾添加百分号。如果程序将输入作为字符串来读取，可将字符串保存在一个 C 字符串变量中，删除任

何不需要的字符，只留一个由数字构成的 C 字符串。然后，程序将这个 C 字符串转换成数字。使用预定义函数 atoi 可轻松实现这一转换。

atoi 函数获取一个 C 字符串参数，返回与之对应的 int 值。例如，atoi("1234")返回整数 1234。如参数不对应一个 int 值，atoi 返回 0。例如，atoi("#37")返回 0，因为字符'#'并非数字。atoi 要念成“A to I”，是“alphabetic to integer”的缩写。atoi 函数在头文件为 cstdlib 的库中，所以要在程序中使用该函数必须包括以下预编译指令：

```
#include <cstdlib>
```

如数字太大，不能转换成 int 值，可考虑把它们从 C 字符串转换成 long 值。atol 函数可执行与 atoi 函数相似的转换，只是返回 long 值，能支持更大整数。

图 8.2 包含 readAndClean 函数的定义，它读取一行输入，丢弃除'0'～'9'之外的其他所有字符。然后，该函数使用 atoi 函数，将纯粹由数字构成的 C 字符串转换成整数值。如示范程序所示，可用该函数读取货币金额，不用关心用户是否包含了一个美元符号。类似地，也可读取百分比，不用关心用户是否输入了百分号。从输出看，虽然 readAndClean 函数表面上只是删除了一些符号，但实际会发生更多的事情。它生成的是真正的 int 值(而非 C 字符串)，可在程序中作为真正的数字使用。

图 8.2　将 C 字符串转换成整数

```
1   // 演示 readAndClean 函数
2   #include <iostream>
3   #include <cstdlib>
4   #include <cctype>
5
6   void readAndClean(int& n);
7   // 读取一行输入，丢弃除数字之外的所有符号
8   // 将 C 字符串转换成整数，将 n 设置成该整数的值
9
10  void newLine();
11  // 丢弃当前输入行上剩余的所有输入
12  // 同时丢弃行末的'\n'
13
14  int main()
15  {
16      using namespace std;
17      int n;
18      char ans;
19      do
20      {
21          cout << "Enter an integer and press return: ";
22          readAndClean(n);
23          cout << "That string converts to the integer " << n << endl;
24          cout << "Again? (yes/no): ";
25          cin >> ans;
26          newLine();
27      } while ( (ans != 'n') && (ans != 'N') );
28      return 0;
29  }
30  // 使用 iostream，cstdlib 和 cctype:
31  void readAndClean(int& n)
32  {
33      using namespace std;
34      const int ARRAY_SIZE = 6;
35      char digitString[ARRAY_SIZE];
36
37      char next;
38      cin.get(next);
39      int index = 0;
```

```
40      while (next != '\n')
41      {
42          if ( (isdigit(next)) && (index < ARRAY_SIZE - 1) )
43          {
44              digitString[index] = next;
45              index++;
46          }
47          cin.get(next);
48      }
49      digitString[index] = '\0';
50      n = atoi(digitString);
51  }
52  // 使用 iostream:
53  void newLine()
54  {
55      using namespace std;
        <newLine 剩余的定义与图 6.7 相同>
```

示范对话

```
Enter an integer and press Return: $ 100
That string converts to the integer 100
Again? (yes/no): yes
Enter an integer and press Return: 100
That string converts to the integer 100
Again? (yes/no): yes
Enter an integer and press Return: 99%
That string converts to the integer 99
Again? (yes/no): yes
Enter an integer and press Return: 23% &&5 *12
That string converts to the integer 23512
Again? (yes/no): no
```

图 8.2 的 readAndClean 函数从输入字符串中删除所有非数字的字符，但不能检查剩余的字符是否能生成用户想要的数字。应该允许用户检查值，判断是否正确。如果不正确，就让用户重新输入。图 8.3 在名为 getInt 的函数中使用了 readAndClean 函数。getInt 接受用户输入的任何内容，并允许重新输入，直到用户对根据输入字符串而生成的数字感到满意。这是一种可靠的输入例程(getInt 函数是图 6.7 的同名函数的增强版本)。

图 8.2 的 readAndClean 与图 8.3 的 getInt 都将数值输入作为字符串值来读入。本章末尾的编程项目 3 要求定义与 getInt 相似的函数，读入 double 而不是 int 值。为了写这个函数，最好有一个预定义函数能将字符串值转换成 double 值。幸好，在头文件为 cstdlib 的库中，预定义函数 atof 正好能做这件事情。例如，atof("9.99")返回 double 值 9.99。参数对应不了 double 类型的数字，atof 就返回 0.0。atof 要念成“A to F”，是“alphabetic to floating point”的缩写。记住，带小数点的数通常称为浮点数，它反映了计算机将这种数字存储到内存时对小数点的处理方式。

图 8.3　可靠的输入函数

```
1  // 演示 getInt 函数的增强版本(原始版本见图 6.7)
2  #include <iostream>
3  #include <cstdlib>
4  #include <cctype>
5  void readAndClean(int& n);
6  // 读取一行输入，丢弃除数字之外的所有符号
7  // 将 C 字符串转换成整数，并将 n 设置成这个整数的值
8  void newLine();
9  // 丢弃当前输入行上的剩余的所有输入
```

```
10 // 同时丢弃行末的'\n'
11 void getInt(int& inputNumber);
12 // 为 inputNumber 赋一个用户认可的值
13 int main()
14 {
15     using namespace std;
16     int inputNumber;
17     getInt(inputNumber);
18     cout << "Final value read in = " << inputNumber << endl;
19     return 0;
20 }
21 // 使用 iostream 和 readAndClean:
22 void getInt(int& inputNumber)
23 {
24     using namespace std;
25     char ans;
26     do
27     {
28         cout << "Enter input number: ";
29         readAndClean(inputNumber);
30         cout << "You entered " << inputNumber
31              << " Is that correct? (yes/no): ";
32         cin >> ans;
33         newLine();
34     } while ((ans != 'y') && (ans != 'Y'));
35 }
36 // 使用 iostream，cstdlib 和 cctype:
37 void readAndClean(int& n)
   <readAndClean 剩余的定义与图 8.2 相同>
38 // 使用 iostream:
39 void newLine()
   <newLine 剩余的定义与图 8.2 相同>
```

示范对话

```
Enter input number: $57
You entered 57 Is that correct? (yes/no): no
Enter input number: $77*5xa
You entered 775 Is that correct? (yes/no): no
Enter input number: 77
You entered 77 Is that correct? (yes/no): no
Enter input number: $75
You entered 75 Is that correct? (yes/no): yes
Final value read in = 75
```

将 C 字符串转换成数值的函数

atoi，atol 和 atof 函数将数字构成的 C 字符串转换成对应数值。atoi 和 atol 将 C 字符串转换成整数。atoi 和 atol 唯一的区别是 atoi 返回 int 值，而 atol 返回 long 值。atof 将 C 字符串转换成 double 值。如 C 字符串实参无法转换成数字，这些函数会返回 0。例如：

```
int x = atoi("657");
```

将 x 的值设为 657，而

```
double y = atof("12.37");
```

将 y 的值设为 12.37。使用了这些函数的程序必须包含以下预编译指令：

```
#include <cstdlib>
```

8.2　标准 string 类

我试图捕捉每一句话，每一个字，然后迅速将所有这些句子和字收进我的文学库，或许将来有一天派得上用场！

——*契诃夫*[①]*(1860—1940)，《海鸥》*

8.1 节介绍了 C 字符串。这些 C 字符串实际是字符数组，只是最后以空字符'\0'终止。处理这些 C 字符串需了解数组处理的全部细节。例如，要在 C 字符串中添加字符，但数组没有足够空间，就必须创建另一个数组来容纳较长的字符串。总之，C 字符串要求程序员跟踪掌握 C 字符串在内存中存储时的所有详细细节。这要求大量额外的工作，也容易造成程序错误。最新的 ANSI/ISO C++标准规定，C++必须提供一个 string 类，允许程序员将字符串视为基本数据类型，不必关心实现细节。本节将介绍这个 string 类型。

标准类 string 简介

string 类在名称同样为<string>的库中定义，其定义包含在 std 命名空间内。所以，要使用 string 类必须包含以下语句(或其他等价语句)：

```
#include <string>
using namespace std;
```

使用 string 类，可采取与处理简单类型的值相似的方式处理 string 值和 string 表达式。可用操作符=为 string 变量赋值，并用操作符+连接两个字符串。例如，假定 s1、s2 和 s3 都是 string 类型的对象，而且 s1 和 s2 都存储了字符串值，那么使用以下表达式，即可使 s3 等于 s1 和 s2 的连接结果：

```
s3 = s1 + s2;
```

不必担心 s3 太小以至于容不下新的字符串值。假如 s1 和 s2 的长度加起来超过 s3 的容量，会自动为 s3 分配更多空间。

本章前面说过，引号字符串实际是 C 字符串，所以它们并非天生就是 string 类型。不过，C++提供了自动转型机制，可将引号字符串转换成 string 类型的值。所以，可将引号字符串直接当作 string 类型的值来处理，而且我们(以及其他大多数人)都经常将引号字符串看成是 string 类型的值。例如以下语句：

```
s3 = "Hello Mom!";
```

它将 string 变量 s3 的值设置成 string 对象，其中包含的字符与 C 字符串"Hello Mom!"相同。

string 类的默认构造函数能将一个 string 对象初始化成空字符串。string 类还有另一个构造函数，它获取一个标准 C 字符串作为参数，所以参数可以是一个引号字符串。该

① 安东·巴甫洛维奇·契诃夫(Anton chekhov)，俄国小说家、戏剧家、十九世纪末期俄国批判现实主义作家、短篇小说艺术大师。他和法国的莫泊桑与美国的欧·亨利齐名，三大短篇小说巨匠之一。——译注

构造函数将 string 对象初始化成一个值，该值代表与 C 字符串参数相同的一个字符串。例如以下语句：

```
string phrase;
string noun("ants");
```

第一行声明字符串变量 phrase，把它初始化成空字符串。第二行将 noun 声明为 string 类型，并把它初始化成等价于 C 字符串"ants"的字符串值。大多数程序员都把它简单地描述成“noun 初始化成"ants"”，但实际发生的是一次类型转换。引号字符串"ants"是 C 字符串，而不是 string 值。noun 变量获得的 string 值虽然包含与"ants"相同的字符，字符顺序也一样，但 string 值不以空字符'\0'结束。事实上(至少理论上)，你不知道(或者不关心)noun 的 string 值是否存储在一个数组中，这与其他一些数据结构相反。

还有另一种方式在声明 string 变量的同时调用一个构造函数。以下两行等价：

```
string noun("ants");
string noun = "ants";
```

图 8.4 的程序展示了和 string 类有关的这些基本细节。注意，正如这个程序所展示的，可用操作符<<输出 string 值。

图 8.4 使用 string 类的程序

```
1   // 演示标准类 string
2   #include <iostream>
3   #include <string>
4   using namespace std;
5   int main()
6   {
7       string phrase;          ← 初始化为空字串
8       string adjective("fried"), noun("ants");  ← 初始化字符串变量的两种方式
9       string wish = "Bon appetite!";            ←
10      phrase = "I love " + adjective + " " + noun + "!";
11      cout << phrase << endl
12           << wish << endl;
13      return 0;
14  }
```

示范对话

```
I love fried ants!
Bon appetit!
```

以图 8.4 的下面这行代码为例：

```
phrase = "I love " + adjective + " " + noun + "!";
```

为了以这种简单自然的方式连接字符串，C++幕后做了大量工作。字符串常量"I love "不是 string 类型的对象。像"I love "这样的字符串常量作为 C 字符串来存储(换言之，作为空终止的字符数组来存储)。一旦 C++看见作为+的参数使用的"I love "，就会查找适用于"I love "这样的一个值的+的定义(或重载)。操作符+有多个重载版本，既有 C 字符串在左侧，string 在右侧的版本；也有位置刚好相反的版本。甚至有个版本允许在操作符+的两侧都使用 C 字符串，返回值则是一个 string 对象。当然，两个操作数都是 string 的重载版本也有。

C++并非真的需要为操作符+提供所有这些重载版本。假如没有合适的重载，C++会查找一个能执行类型转换的构造函数，将 C 字符串"I love "转换成适合使用操作符+的一个值。在本例中，支持一个 C 字符串参数的构造函数恰好能执行这种转换。然而，如果事先进行了额外的重载，那么或许效率更高(因为不需要转换)。

通常将 string 类视为 C 字符串的一种新型替代物。但在 C++中使用 string 类来编程时，不可能完全做到杜绝 C 字符串。

> **string 类**
>
> string 类用于表示字符串值，该类提供了比 8.1 节讨论的 C 字符串更全面的字符串表示和处理功能。
>
> string 类在名称同为<string>的库中定义，定义放在 std 命名空间中。所以，要使用 string 类，代码必须包含以下语句(或其他等价的语句)：
>
> ```
> #include <string>
> using namespace std;
> ```
>
> string 类的默认构造函数将 string 对象初始化成空字符串，另一个构造函数获取一个 C 字符串作为参数，将 string 对象初始化成一个值来表示由参数给定的字符串。例如：
>
> ```
> string s1, s2("Hello");
> ```

string 类的 I/O

可用插入操作符<<和 cout 来输出 string 对象，这和处理其他类型的数据是一样的。图 8.4 已对此进行了展示。string 类的输入则稍微麻烦一些。

为 string 对象使用提取操作符>>和 cin 时，方式与其他数据一样。但要记住，提取操作符会忽略最初的空白字符，并在遇到更多的空白字符时停止读入。这一点对于字符串和其他数据来说都是成立的。例如以下代码：

```
string s1, s2;
cin >> s1;
cin >> s2;
```

如果用户输入：

```
May the hair on your toes grow long and curly!
```

那么 s1 获得的值是"May"，同时删除前置(或尾随)的所有空白字符。变量 s2 获得的值是"the"。使用提取操作符>>和 cin 只能读取单词，不能读取整行或包含一个空格的其他字符串。这有时是你希望的行为，有时则不是。

使用 getline 函数将整行内容读入 string 类型的变量。为 string 对象使用 getline 函数时，语法稍微有别于用于 C 字符串时的语法(参见 8.1 节)。不能使用 cin.getline；相反，要将 cin 作为 getline 的第一个参数[①](因此，这个版本的 getline 不是成员函数)。

```
string line;
```

① 这有一定的讽刺意味。string 类是采用现代的面向对象技术设计的，getline 采用的却是一种古老的语法形式，不是面向对象的形式。这要归咎于历史上的一次意外。这个 getline 函数是在 iostream 库投入使用之后才设计的。设计者没有太多的选择，无奈之下只好将这个 getline 设计成一个独立函数。

```
cout << "Enter a line of input:\n";
getline(cin, line);
cout << line << "END OF OUTPUT\n";
```

将它们嵌入一个完整的程序，上述代码会生成如下所示的对话：

```
Enter some input:
Do be do to you!
Do be do to you!END OF OUTPUT
```

如行中有前置或尾随空格，它们也会成为 getline 读入的字符串值的一部分。这个版本的 getline 在<string>库中定义，可将 cin 替换成与文本文件连接的流对象，让 getline 从文件中输入数据。

不能用 cin 和>>读入空白字符。要一次读入一个字符，可使用 cin.get，它的详情已在第 6 章讨论。cin.get 函数读取的是 char 类型的值，而不是 string 类型的值。但在处理 string 输入时，该函数能提供一定的帮助。图 8.5 的程序展示了在进行 string 输入时同时使用 getline 和 cin.get 的情况。newLine 函数的重要性在稍后的“陷阱：混合使用 cin>>*变量*;和 getline”一节讲述。

图 8.5　使用了 string 类的程序

```
 1  // 演示 getline 和 cin.get
 2  #include <iostream>
 3  #include <string>

 4  void newLine();

 5  int main()
 6  {
 7      using namespace std;
 8
 9      string firstName, lastName, recordName;
10      string motto = "Your records are our records.";

11      cout << "Enter your first and last name:\n";
12      cin >> firstName >> lastName;
13      newLine();

14      recordName = lastName + ", " + firstName;
15      cout << "Your name in our records is: ";
16      cout << recordName << endl;

17      cout << "Our motto is\n"
18           << motto << endl;
19      cout << "Please suggest a better (one line) motto:\n";
20      getline(cin, motto);
21      cout << "Our new motto will be:\n";
22      cout << motto << endl;

23      return 0;
24  }
25
26  // 使用 iostream:
27  void newLine()
28  {
29      using namespace std;
30
31      char nextChar;
32      do
33      {
```

```
34        cin.get(nextChar);
35     } while (nextChar != '\n');
36  }
```

示范对话

```
Enter your first and last name:
B'Elanna Torres
Your name in our records is: Torres, B'Elanna
Our motto is
Your records are our records.
Please suggest a better (one-line) motto:
Our records go where no records dared to go before.
Our new motto will be:
Our records go where no records dared to go before.
```

string 对象的 I/O

插入操作符<<与 cout 配合使用可输出 string 对象。提取操作符>>和 cin 配合可输入 string。用>>输入时，代码读取的是以空白字符定界的字符串。可用 getline 函数将整行文本输入 string 对象。

示例

```
string greeting("Hello"), response, nextWord;
cout << greeting << endl;
getline(cin, response);
cin >> nextWord;
```

自测题

15. 给定以下代码(假定嵌入一个完整而正确的程序后再运行)：

```
string s1, s2;
cout << "Enter a line of input:\n";
cin >> s1 >> s2;
cout << s1 << "*" << s2 << "<END OF OUTPUT";
```

如果对话像下面这样开始，下一行输出会是什么？

```
Enter a line of input:
A string is a joy forever!
```

16. 给定以下代码(假定嵌入一个完整而正确的程序后再运行)：

```
string s;
cout << "Enter a line of input:\n";
getline(cin, s);
cout << s << "<END OF OUTPUT";
```

如果对话像下面这样开始，下一行输出会是什么？

```
Enter a line of input:
A string is a joy forever!
```

编程提示：getline 的其他版本

通过前面的学习，我们知道 getline 可以像下面这样使用：

```
string line;
cout << "Enter a line of input:\n";
getline(cin, line);
```

这个版本在遇到行结束符'\n'时停止读入。还有一个版本允许将其他字符指定为停止信号。例如，以下语句在遇到第一个问号时停止读入：

```
string line;
cout << "Enter a line of input:\n";
getline(cin, line, '?');
```

可在使用 getline 时把它视为 void 函数，但它实际会返回对其第一个参数的引用。在上述代码中，第一个参数是 cin。所以，以下语句将一行文本读入 s1，将不含空白字符的字符串读入 s2：

```
string s1, s2;
getline(cin, s1) >> s2;
```

getline(cin, s1)返回对 cin 的引用，所以调用 getline 之后，接着发生的事情就等价于执行以下代码：

```
cin >> s2;
```

getline 的这种用法似乎是为 C++的课堂测验而设计的，而不是为了满足任何实际的编程需求。但某些时候，它仍有可能带来一定的便利。■

陷阱：混合使用 cin>>变量;和 getline

视频讲解：*Example using cin and getline with the string class*

混合使用 cin >> *变量*;和 getline 进行输入务必谨慎。例如以下代码：

```
int n;
string line;
cin >> n;
getline(cin, line);
```

> **string 类的 getline 成员函数**
>
> string 类的 getline 成员函数有以下两个版本：
>
> ```
> istream& getline(istream& ins, string& strVar, char delimiter);
> istream& getline(istream& ins, string& strVar);
> ```
>
> 第一个版本从作为第一个参数给定的 istream 对象(本章肯定是 cin)中读入字符，将其插入 string 变量 strVar 中，直到遇到 delimiter 字符(定界符)的一个实例。delimiter 字符会从输入中删除并丢弃。相较于第一个版本，第二个版本唯一的变化就是将'\n'作为 delimiter 的默认值使用。
>
> 这两个 getline 函数都返回它们的第一个参数(本章肯定是 cin)，但通常可将它们视为 void 函数。

如提供以下输入，你可能以为 n 的值会被设为 42，line 的值会被设为代表"Hello hitchhiker."的字符串：

```
42
Hello hitchhiker.
```

虽然 n 值确实会被设为 42，但 line 最终存储的是一个空字符串。为什么？

使用 cin >> n 时，会忽略输入中的前置空白字符，但会将行内剩下的内容(本例只剩下'\n')留作下一个输入。对于以下形式的语句：

```
cin >> n;
```

它总是将行内剩下的东西留给后续的 getline 来读取(即使只剩下'\n')。在本例中，getline 看到'\n'后停止读入，所以 getline 实际读入空字符串。如程序莫名其妙地忽略输入数据，请检查是否混合使用了这两种输入。可能需要使用图 8.5 的 newLine 函数或者来自 iostream 库的 ignore 函数。例如以下语句：

```
cin.ignore(1000, '\n');
```

用这些参数调用 ignore 成员函数，函数会一边读取，一边丢弃字符，直到遇到'\n'。如读取并丢弃 1000 个字符之后还是没有遇到'\n'，就只丢弃这 1000 个字符。

在一个程序中，如果同时为>>和 getline 使用了 cin，还有可能遇到其他令人困惑的问题。除此之外，在不同的 C++编译器上，这些问题可能会有不同的表现。如果你的其他办法都行不通，或者想保证移植性，可考虑使用 cin.get 来依次输入每个字符。

本章讨论的任何版本的 getline 均可能发生这些问题。 ■

用 string 类进行字符串处理

string 类允许进行各种字符串处理。除了 8.1 节提到的针对 C 字符串的操作，它还支持其他操作。可采取与访问数组元素一样的方式访问 string 对象中的字符。所以，string 对象除了具有字符数组的一切优势，还具有数组没有的优势，比如自动扩容能力。对于名为 lastName 的 string 对象，lastName[i]能访问 lastName 所代表的字符串中的编号为 i 的字符。图 8.6 展示了数组方括号的这种用法。

图 8.6 还演示了成员函数 length 的用法。每个 string 对象都有名为 length 的成员函数，它不获取任何参数，返回 string 对象所代表的那个字符串的长度。所以，一个 string 对象不仅能像数组那样使用，还能借助于成员函数 length，像一个部分填充的数组那样，自动跟踪实际使用的存储位置数目。

图 8.6 string 对象可以具有类似于数组的行为

```
1  // 像使用数组一样使用 string 对象
2  #include <iostream>
3  #include <string>
4  using namespace std;
5  int main()
6  {
7      string firstName, lastName;
8      cout << "Enter your first and last name:\n";
9      cin >> firstName >> lastName;
10     cout << "Your last name is spelled:\n";
11     int i;
12     for (i = 0; i < lastName.length(); i++)
13     {
14         cout << lastName[i] << " ";
15         lastName[i] = '-';
16     }
17     cout << endl;
18     for (i = 0; i < lastName.length(); i++)
```

```
19          cout << lastName[i] << " "; // 在每个字母下显示一个"-"
20      cout << endl;
21      cout << "Good day " << firstName << endl;
22      return 0;
23  }
```

示范对话

```
Enter your first and last name:
John Crichton
Your last name is spelled:
C r i c h t o n
- - - - - - - -
Good day John
```

数组方括号用于 string 对象时不检查非法索引。如使用了非法索引(也就是说，索引值大于或等于对象中的字符串的长度)，会产生无法预料的结果(而且通常是不好的结果)。程序可能产生奇怪的行为，而且没有任何错误消息指出问题出在一个非法索引值上。

有一个名为 at 的成员函数能检查非法索引值。at 的行为与方括号相同，除了以下两点：at 是函数，所以不要写成 a[i]，要写成 a.at(i)；另外，at 函数会检查 i 是否求值为非法索引。如 a.at(i)中的 i 值是非法索引，会生成一条运行时错误消息，指出具体错在哪里。后面的两段示范代码试图进行越界访问。但第一段代码也许不会产生错误消息，即使它访问不存在的索引变量：

```
string str("Mary");
cout << str[6] << endl;
```

但第二段代码会导致程序异常终止，所以你至少知道其中存在错误：

```
string str("Mary");
cout << str.at(6) << endl;
```

但要注意，某些系统在发现 str.at(i)具有非法索引 i 的时候会给出模棱两可的错误消息。

为更改字符串中的单个字符，可为索引变量(比如 str[i])赋一个 char 值。另外，还可使用成员函数 at 做这件事情。例如，要将 string 对象 str 中的第 3 个字符更改为'x'，可以使用以下任何一个语句：

```
str.at(2) = 'X';
```

或者：

```
str[2] = 'X';
```

和普通字符数组一样，string 对象的字符位置也从 0 开始编号，所以字符串中的第 3 个字符的索引编号是 2。

图 8.7 总结了 string 类的部分成员函数。

string 类的对象在许多方面都具有比 C 字符串(参见 8.1 节)更好的行为。为 string 类的对象使用操作符==，会返回更人性化的结果。也就是说，如果两个字符串包含相同的字符，而且所有字符顺序相同，那么返回 true；否则返回 false。类似地，比较操作符<，>，<=和>=使用词典顺序比较 string 对象。使用词典顺序时，实际使用的是附录 3 给出的 ASCII 字符的顺序。如字符串只由字母构成，而且字母全部大写或小写，那么词典顺序就等于字母顺序。

图 8.7　string 标准类的成员函数

示例	说明
构造函数	
string str;	默认构造函数，创建名为 str 的空 string 对象
string str("sample");	创建 string 对象，其中含有数据"sample"
string str(aString);	创建名为 str 的 string 对象，它是 aString 的拷贝；aString 是 string 类的对象
元素访问	
str[i]	返回对 str 中的索引 i 处的字符的读/写引用。不检查非法索引
str.at(i)	返回对 str 中的索引 i 处的字符的读/写引用，这一点和 str[i]相同，但这个版本会检查非法索引
str.substr(position, length)	返回调用对象(也就是 str)的一个子字符串。子串始于 position，含有 length 个字符
str.length()	返回 str 的长度
赋值/修改	
str1 = str2;	将 str1 初始化成 str2 的数据
str1 += str2;	str2 的字符数据连接到 str1 的尾部
str.empty()	如果 str 是空字符串，就返回 true；否则返回 false
str1 + str2	返回将 str2 数据连接到 str1 数据尾部的字符串
str.insert(pos, str2);	在 str 的 pos 处插入 str2
str.erase(pos, length);	删除长度为 length 的子字符串，从 pos 处开始
比较	
str1 == str2　str1 != str2	比较相等还是不相等；返回布尔值
str1 < str2　str1 > str2	4 个比较都基于词典顺序
str1 <= str2　str1 >= str2	
查找	
str.find(str1)	返回 str1 在 str 中首次出现的索引。如果 str1 没有找到，返回特殊值 string::npos
str.find(str1, pos)	返回 str1 在 str 中首次出现的索引，从 pos 处开始查找
str.find_first_of(str1, pos)	返回 str1 的任何字符在 str 中首次出现的索引，从 pos 处开始查找
str.find_first_not_of(str1, pos)	返回不属于 str1 的任何字符在 str 中首次出现时的索引，从 pos 处开始查找

对于 string 对象和 C 字符串，操作符=和==具有不同的行为

将操作符=，==，!=，<，>，<=和>=用于 C++标准 string 类型时，结果与我们期望的相符。用于 C 字符串时，却可能产生较为别扭的行为(参见 8.1 节)。

编程实例　回文测试

回文是顺读和倒读都一样的字符串。图 8.8 的程序测试输入字符串是不是回文。该回文测试忽略所有空格和标点符号，并忽略字母的大写和小写形式。下面是回文的一些例子：

```
Able was I ere I saw Elba.
I Love Me, Vol. I.
Madam, I'm Adam.
A man, a plan, a canal, Panama.
Rat live on no evil star.
radar
deed
mom
racecar
```

removePunct 函数有趣的地方在于使用了 string 的成员函数 substr 和 find。substr 提取调用对象的一个子字符串，该子字符串从给定位置开始截取，具有指定长度。

图 8.8　回文测试程序

```
1  // 测试是否回文
2  #include <iostream>
3  #include <string>
4  #include <cctype>
5  using namespace std;

6  void swap(char& v1, char& v2);
7  // 交换 v1 和 v2 的值

8  string reverse(const string& s);
9  // 返回 s 的拷贝，但反转字符顺序

10 string removePunct(const string& s, const string& punct);
11 // 返回 s 的拷贝，同时删除字符串 punct 中的任何字符的任何实例

12 string makeLower(const string& s);
13 // 返回 s 的拷贝，将所有大写形式变成小写形式，其他字符不变

14 bool isPal(const string& s);
15 // 是回文就返回 true，否则返回 false

16 int main()
17 {
18    string str;
19    cout << "Enter a candidate for palindrome test\n"
20         << "followed by pressing return.\n";
21    getline(cin, str);

22    if (isPal(str))
23        cout << "\"" << str + "\" is a palindrome.";
24    else
25        cout << "\"" << str + "\" is not a palindrome.";
26    cout << endl;

27    return 0;
28 }
29
30 void swap(char& v1, char& v2)
31 {
32    char temp = v1;
33    v1 = v2;
34    v2 = temp;
35 }
36
37 string reverse(const string& s)
38 {
39    int start = 0;
40    int end = s.length();
41    string temp(s);
42
43    while (start < end)
44    {
45       end--;
46       swap(temp[start], temp[end]);
47       start++;
48    }
```

```
49     return temp;
50 }
51 // 使用<cctype>和<string>
52 string makeLower(const string& s)
53 {
54     string temp(s);
55     for (int i = 0; i < s.length(); i++)
56         temp[i] = tolower(s[i]);
57     return temp;
58 }
59 string removePunct(const string& s, const string& punct)
60 {
61     string noPunct; //初始化成空字符串
62     int sLength = s.length();
63     int punctLength = punct.length();
64     for (int i = 0; i < sLength; i++)
65     {
66         string aChar = s.substr(i,1); //一个单字符 string
67         int location = punct.find(aChar, 0);
68         // 查找 aChar 在 punct 中出现的位置
69       if (location < 0 || location >= punctLength)
70         noPunct = noPunct + aChar; // 没有在 punct 中找到 aChar，所以保留 aChar
71     }
72     return noPunct;
73 }
74 // 使用函数 makeLower，removePunct
75 bool isPal(const string& s)
76 {
77     string punct(",;:.?!'\" "); // 有一个空格
78     string str(s);
79     str = makeLower(str);
80     string lowerStr = removePunct(str, punct);
81     return (lowerStr == reverse(lowerStr));
82 }
```

示范对话 1

```
Enter a candidate for palindrome test
followed by pressing Return.
Madam, I'm Adam.

"Madam, I'm Adam." is a palindrome.
```

示范对话 2

```
Enter a candidate for palindrome test
followed by pressing Return.
Radar

"Radar" is a palindrome.
```

示范对话 3

```
Enter a candidate for palindrome test
followed by pressing Return.
Am I a palindrome?

"Am I a palindrome?" is not a palindrome.
```

removePunct 函数前三行声明要在函数中使用的变量。for 循环每次处理参数 s 的一个字符，尝试用 find 函数在 punct 字符串中查找它们。为此，会从 s 的每个字符位置提取长度为 1 的一个子字符串。该子字符串在 punct 字符串中的位置用 find 成员函数确定。如果由单个字符构成的这个字符串不在 punct 字符串中，就将这个单字符的 string 连接到准备返回的 noPunct 字符串上。

√ 自测题

17. 给定以下代码：

```
string s1, s2("Hello");
cout << "Enter a line of input:\n";
cin >> s1;
if (s1 == s2)
    cout << "Equal\n";
else
    cout << "Not equal\n";
```

如果对话像下面这样开始，下一行输出会是什么？

```
Enter a line of input:
Hello friend!
```

18. 以下代码的输出是什么？

```
string s1, s2("Hello");
s1 = s2;
s2[0] = 'J';
cout << s1 << " " << s2;
```

string 对象和 C 字符串之间的转换

前面说过，C++能执行自动类型转换，所以可将 C 字符串存储在 string 类型的变量中。例如，以下代码能很好地工作：

```
char aCString[] = "This is my C string.";
string stringVariable;
stringVariable = aCString;
```

但是，以下代码会使编译器报错：

```
aCString = stringVariable; //非法
```

以下代码同样非法：

```
strcpy(aCString, stringVariable); //非法
```

strcpy 不允许 string 对象作为它的第二个参数，而且 string 对象不能自动转换为 C 字符串，这是无法逃避的一个问题。

为了获得和 string 对象对应的 C 字符串，必须执行显式类型转换。这要用到 string 成员函数 c_str()。为了用 strcpy 复制字符串，正确的版本如下：

```
strcpy(aCString, stringVariable.c_str()); //合法
```

注意，复制要用 strcpy 函数进行。成员函数 c_str()返回与 string 调用对象对应的 C 字符串。本章前面讲过，赋值操作符不适用于 C 字符串。所以，以下看似合法的语句其

实也是非法的：

```
aCString = stringVariable.c_str(); //非法
```

字符串和数字之间的转换

C++11 之前，字符串和数字之间的转换有点复杂，但 C++11 就是调用一个函数的事情。stof，stod，stoi 和 stol 分别将字符串转换成 float，double，int 和 long。to_string 将数值类型转换成字符串。下例演示了这些函数：

```
int i;
double d;
string s;
i = stoi("35"); // 字符串"35"转换成整数 35
d = stod("2.5"); // 字符串"2.5"转换成 double 值 2.5
s = to_string(d*2); // double 值 5.0 转换成字符串"5.0000"
cout << i << " " << d << " " << s << endl;
```

输出结果如下：

```
35 2.5 5.0000。
```

8.3 向　　量

"好，我就吃它，"爱丽丝说，"如果它使我变大，我就可以够得着钥匙了；如果它使我变得更小，我就可以从门缝下面爬过去。反正不管怎样，我都可以到那个花园里去了……"

——*刘易斯·卡洛尔，《爱丽丝漫游奇境记》*

可将向量(vector)视为能在程序运行时改变长度(增大或缩小)的数组。C++数组一旦创建就不能改变长度。向量作用和数组一样，只是长度能在程序运行时自由改变。向量是 STL(Standard Template Library，标准模板库)的一部分，后者是一个 C++标准库，详情将在第 18 章讨论。

学习本节内容之前，不需要先阅读本章之前的各个小节。

向量基础知识

类似于数组，向量也有基类型，而且存储的同样是基类型的一组值。但向量类型和向量变量的声明语法有别于数组。

以下语句将变量 v 声明为基类型为 int 的向量：

```
vector<int> v;
```

vector<*Base_Type*>代表一个**模板类**，可将 *Base_Type* 替换成任何类型(这里是 int)，最终为那种基类型的向量生成一个类。为向量指定基类型时，基本原理和为数组指定基类型一样。任何类型(包括类类型)都可作为向量的基类型。vector<int>是类名，所以上述语句将 v 声明为 vector<int>类型的向量时，包括了对 vector<int>类的默认构造函数的一个调用。该构造函数创建一个空向量对象(无元素)。

和数组一样，向量元素索引编号从 0 开始。可通过数组方括号记号法读取或更改元素，这也和数组一样。例如，以下代码更改向量 v 的编号为 i 的元素，输出更改过的值(i 是 int 变量)：

```
v[i] = 42;
cout << "The answer is " << v[i];
```

但将方括号用于向量时存在一项限制。虽然可用 v[i]更改编号为 i 的元素的值，但不能使用 v[i]初始化编号为 i 的元素。只能用它更改已赋值的元素。在向量索引位置首次添加元素时，通常要使用成员函数push_back。

在向量中添加元素时，要依次添加，首先添加位置 0，然后添加位置 1，位置 2，依此类推。成员函数 push_back 在下一个可用的位置添加一个元素。例如，以下代码为向量 sample 的元素 0，1 和 2 添加初始值：

```
vector<double> sample;
sample.push_back(0.0);
sample.push_back(1.1);
sample.push_back(2.2);
```

C++11 允许使用和初始化数组一样的方式初始化向量：

```
vector <double> sample = {0.0, 1.1, 2.2};
```

向量元素个数称为向量的**长度**[①]。成员函数 size 判断向量有多少个元素。例如，在执行上述代码之后，sample.size()返回 3。可像下面这样输出向量 sample 的所有元素：

```
for (int i = 0; i < sample.size(); i++)
    cout << sample[i] << endl;
```

size 函数返回 unsigned int 类型的值，而不是 int 类型(unsigned int 只允许非负整数值)。在需要 int 的地方，它应该能自动转换成 int，但有的编译器会警告你：*在需要 int 的地方使用了一个 unsigned int*。要完全保险可执行强制类型转换，将返回的 unsigned int 转换成 int。另外，也可使用 unsigned int 类型的循环控制变量，如以下 for 循环所示：

```
for (unsigned int i = 0; i < sample.size(); i++)
    cout << sample[i] << endl;
```

还可使用基于范围的 for 循环：

```
for (auto i : sample)
cout << i << endl;
```

图 8.9 演示了一些基本的向量技术。

图 8.9　使用向量

```
1  #include <iostream>
2  #include <vector>
3  using namespace std;

4  int main()
5  {
6      vector<int> v;
7      cout << "Enter a list of positive numbers.\n"
8           << "Place a negative number at the end.\n";
```

① size，而不是 length，本书都翻译成“长度”。——译注

```
 9      int next;
10      cin >> next;
11      while (next > 0)
12      {
13          v.push_back(next);
14          cout << next << " added. ";
15          cout << "v.size() = " << v.size() << endl;
16          cin >> next;
17      }

18      cout << "You entered:\n";
19      for (unsigned int i = 0; i < v.size(); i++)
20          cout << v[i] << " ";
21      cout << endl;

22      return 0;
23  }
```

示范对话

```
Enter a list of positive numbers.
Place a negative number at the end.
2 4 6 8 -1
2 added. v.size() = 1
4 added. v.size() = 2
6 added. v.size() = 3
8 added. v.size() = 4
You entered:
2 4 6 8
```

向量的一个构造函数能获取一个整数参数，初始化由参数指定的位置数目。例如，假定像下面这样声明 v：

```
vector<int> v(10);
```

那么前 10 个元素会初始化成 0，而 v.size()会返回 10。然后，可以让 i 从 0 变化到 9，并用 v[i]为编号为 i 的元素设置值。例如，以下语句可以紧跟在声明语句之后：

```
for (unsigned int i = 0; i < 10; i++)
    v[i] = i;
```

不过，在 i 大于或等于 10 的情况下，要设置编号为 i 的元素，就要用 push_back 了。

向构造函数传递一个整数实参，向量会初始化成相应数据类型的零值。如向量基类型是类，就用那个类的默认构造函数初始化。

向量定义包含在 vector 库中，被放到 std 命名空间。所以，在程序中使用向量需包括以下语句(或其他类似语句)：

```
#include <vector>
using namespace std;
```

向　　量

向量用法类似于数组，但向量长度不固定。如需更大容量来存储更多元素，它的容量就会自动扩充。向量在<vector>库中定义，位于 std 命名空间。所以，在程序中使用向量需包括以下语句(或其他类似语句)：

```
#include <vector>
using namespace std;
```

> 给定了 *Base_Type*(基类型)的向量类要写成 vector<*Base_Type*>。下面是两个示范向量声明：
>
> ```
> vector<int> v;        // 默认构造函数生成一个空向量
> vector<AClass> record(20);   //用 AClass 类的默认构造函数初始化 20 个元素
> ```
>
> 在向量添加元素要用成员函数 push_back，如下例所示：
>
> ```
> v.push_back(42);
> ```
>
> 一个元素位置获得了它的第一个值之后(无论是通过 push_back，还是通过构造函数来初始化)，以后就可使用方括号记号法来访问那个元素位置，和访问普通数组元素一样。

陷阱：使用方括号时超出向量长度

如 v 是向量，而 i 大于或等于 v.size()，则元素 v[i]不存在，必须用 push_back 从位置 i 处添加更多的元素。i 大于或等于 v.size()时，如试图设置 v[i]，比如：

```
v[i] = n;
```

虽然不一定会产生错误消息，但程序肯定会在某个时候出现异常。 ■

编程提示：向量赋值具有良好行为

为向量使用赋值操作符，会逐个元素为赋值操作符左侧的向量赋值(并根据需要扩充容量，以及重置赋值操作符左侧向量的长度)。所以，只要作用于基类型的赋值操作符能生成基类型元素的一个独立拷贝，作用于向量的赋值操作符也能生成(向量的)一个独立拷贝。

注意，赋值操作符为了生成赋值操作符左侧向量的独立拷贝，要求作用于基类型的赋值操作符能生成独立拷贝。为向量使用赋值操作符的结果如何，完全取决于作用于它的基类型的赋值操作符(为类重载赋值操作符的详情将在第 11 章讲述)。 ■

效率问题

向量在任何时候都有一个**容量**[①]，即当前分配了内存的元素的数量。成员函数 capacity()返回向量的容量。不要混淆向量的容量和长度。长度是向量中元素的个数，容量是实际分配了内存的元素的个数。容量通常大于长度，且肯定大于或等于长度。

一旦向量容量不够，并需要空间来容纳一个附加成员，容量就会自动增加。不同 C++ 实现对每次增加的容量有不同规定，但通常都会超过马上就要用到的容量。一个常见的方案是倍增当前需要的容量。由于增加容量是一项复杂的任务，所以相较于频繁分配许多小的内存块，一次性分配一个较大的内存块显得更有效率。

> **长度和容量**
>
> 向量的**长度**是指向量中的元素个数，**容量**是当前实际分配了内存的元素的个数。对于向量 v，可分别用成员函数 v.size()和 v.capacity()判断其长度和容量。

① capacity，而不是 size 或 length。——译注

一般可忽略向量的容量，这不会对程序行为产生任何影响。但如果必须考虑效率问题，可考虑自己管理容量，而不是接受每次都使容量倍增的默认行为。可用成员函数 reserve 来显式增大向量的容量。例如以下语句将容量设为至少 32 个元素：

```
v.reserve(32);
```

以下语句将容量设为向量当前元素个数加 10：

```
v.reserve(v.size() + 10);
```

注意，虽然可以用 v.reserve 增大向量的容量，但在参数值小于当前容量的前提下，它**不一定**减小向量的容量。

成员函数 resize 用于改变向量长度(注意是长度)。例如，以下语句将向量的长度变成 24 个元素：

```
v.resize(24);
```

如原来长度小于 24，新元素会由接收整数参数的构造函数进行初始化。如原来长度大于 24，那么除了前 24 个元素，其他元素都会丢失。如有必要，容量会自动增加。使用 resize 和 reserve，可在不需要部分元素或部分容量的前提下，主动缩小向量的长度和容量。

√ 自测题

19. 以下程序合法吗？如果合法，会输出什么？

```
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    vector<int> v(10);
    int i;

    for (i = 0; i < v.size(); i++)
        v[i] = i;

    vector<int> copy;
    copy = v;
    v[0] = 42;

    for (i = 0; i < copy.size(); i++)
        cout << copy[i] << " ";
    cout << endl;
    return 0;
}
```

20. 向量的长度(size)和容量(capacity)有何区别？

小　结

- C 字符串变量与字符数组相同，只是在使用方式上稍有区别。字符串变量使用空字符'\0'标记数组中存储的一个字符串的结束。
- C 字符串变量通常必须当作数组来处理，而不是当作为数字和字符使用的那种简单变量。尤其要注意的是，不能使用等号(=)将 C 字符串值赋给 C 字符串变量。另外，也不能使用操作符==比较两个 C 字符串变量中的值。相反，必须使用特殊的 C 字符串函数执行这些任务。
- ANSI/ISO 标准<string>库提供了一个全功能的 string 类，可用它以自然的方式表示和处理字符串。
- 和 C 字符串相比，string 类的对象具有更好的行为。特别是，为 string 类的对象使用=和==操作符时，能获得符合希望的结果。
- 向量可被视为长度能在程序运行时自由增大(或缩小)的数组。

自测题答案

1. 以下两个声明等价(但不等价于其他任何声明):

```
char stringVar[10] = "Hello";
char stringVar[10] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

以下两个声明也等价(但不等价于其他任何声明):

```
char stringVar[6] = "Hello";
char stringVar[] = "Hello";
```

以下声明不等价于其他任何声明:

```
char stringVar[10] = {'H', 'e', 'l', 'l', 'o'};
```

2. "DoBeDo to you"

3. 通过声明可以看出，stringVar 只有存储 6 个字符的空间(含空字符'\0')。strcat 函数不检查是否有足够空间在 stringVar 中添加更多的字符，所以 strcat 会将字符串" and Good-bye."中的所有字符都写到内存，即使当前分配给 stringVar 的内存不足以容下这么多字符。最终，不应改动的内存被改动了。结果无法预料，但通常都不好。

4. 如果还没有为你定义 strlen，可以使用以下定义:

```
int strlen(const char str[])
// 前条件: str 包含以'\0'终止的一个字符串值
// 返回字符串 str 中的字符数(不包括'\0')
{
    int index = 0;
    while (str[index] != '\0')
        index++;
    return index;
}
```

5. 最大字符数是 5，因为第 6 个位置要被空终止符'\0'占用。

6. a. 1
   b. 1
   c. 5(包括'\0')
   d. 2(包括'\0')
   e. 6(包括'\0')

7. 两者不等价。第一个将空字符'\0'放在数组中的字符'a'，'b'和'c'之后；第二个只为连续的位置赋值'a'，'b'和'c'，但是不会在任何地方插入一个'\0'。

8.
```
int index = 0;
while ( ourString[index] != '\0' )
{
    ourString[index] = 'X';
    index++;
}
```

9. a. 如 C 字符串变量不包含空终止符'\0'，循环会超出为 C 字符串变量分配的内存范围，从而破坏不属于该变量的内存。为保护数组结尾之后的内存，要像(b)那样更改 while 循环条件。

   b. `while (ourString[index] != '\0' && index < SIZE)`

10.
```
#include <cstring>  // 获得 strcpy 的声明所需
 . . .
strcpy(aString, "Hello");
```

11. I did it my way!

12. 字符串"Good, I hope."对于 aString 来说太长了。不属于 aString 数组的一部分内存会被覆盖。

13.
```
Enter some input:
The
   time is now.
The-timeEND OF OUTPUT
```

14. 完整对话如下：

```
Enter a line of input:
May the hair on your toes grow long and curly.
May t<END OF OUTPUT
```

15. `A*string<END OF OUTPUT`

16. `A string is a joy forever!<END OF OUTPUT`

17. 完整对话如下：

```
Enter a line of input:
Hello friend!
Equal
```

记住，cin 在遇到空白字符(比如空格)时停止读取。

18. `Hello Jello`

19. 程序合法，输出如下：

```
0 1 2 3 4 5 6 7 8 9
```

注意，更改 v 不会更改 copy。使用以下赋值语句，生成了一个真正的独立拷贝：

```
copy = v;
```

20. 长度是向量中的元素数目，容量是实际分配了内存的元素数目。容量通常大于长度。

编 程 练 习

编程练习一般只需写很小的程序，运用本章提到的编程概念。

1. 创建包含名字、年龄和职位的 C 字符串变量。每个字段以空格分隔，例如"Bob 45 Programmer"，假定名字、年龄和职位本身不含空格。写程序只使用来自 cstring 的函数(不要使用 string 类的)将名字、年龄和职位提取到单独的变量中。用多个名字、年龄和职位组合测试程序。
2. 重做编程练习 1，这次使用 string 类提取字段而不用 ctring 的函数。
3. 写程序将以空格分隔的名字和姓氏输入一个字符串变量。使用 string 的函数输出它们的首字母。将代码嵌入 do-while 循环。在循环结束时询问是否想要重复。用 cin 将用户的选择输入一个 char。'y'重复程序，否则退出。注意 cin 和 getline 混合使用时换行符可能造成的问题。
4. 写 firstLast2 函数获取一个整数向量。向量以 2 开头或结尾就返回 true，否则返回 false。使用多个向量测试函数，要求这些向量具有不同长度，2 出现在向量开头和结尾，出现在中间，或者完全没有 2。
5. 写 swapFrontBack 函数获取一个整数向量。函数交换第一个和最后一个元素。要检查向量是否为空以防止出错。使用具有不同长度和前后数字的多个向量测试函数。
6. 重做编程练习 7.4，修改程序使用字符串向量而不是字符串数组。
7. 写程序输入两个字符串变量 first 和 last，分别代表用户的名字和姓氏。首先将两个字符串转换成全部小写的形式。然后创建一个新字符串，用 pig latin 格式来编码用户的全名。编码后的名字和姓氏要求首字母大写。单词编码成 pig latin 格式的规则如下。

   (1) 首字母是辅音字母，移动到末尾，在末尾添加“ay”。

(2) 首字母是元音字母，在末尾添加“way”。

例如，假定输入“Erin”作为名字，输入“Jones”作为姓氏，程序应创建新字符串“Erinway Onesjay”并输出。

编 程 项 目

编程项目要求综合运用多方面的知识来解决问题，程序一般比编程练习大，解题方式多样化。

1. 视频讲解：*Solution to Programming Project 8.1*

   写程序读入一个最长 100 个字符的句子。对这个句子进行调整，输出间距正确，且首字母正确大写的结果。换言之，在输出句子中，由两个或者更多空格构成的所有字符串都必须压缩成一个空格。句首字母必须大写，但除此之外不能再有其他任何大写字母。不要关心专有名称，允许把它们的首字母变成小写。将换行符视为空格，需要将任意数量的换行符压缩成单独一个空格。假定句子以句点结尾，而且句子中没有其他句点。例如，以下输入：

```
the   Answer to life, the Universe, and   everything
IS 42.
```

   所生成的输出如下：

```
The answer to life, the universe, and everything is 42.
```

2. 写程序读入一行文本，输出这一行文本的单词数目以及每个字母出现的次数。将单词定义为一个字母字符串，位于它两端的定界符可以是一个空白字符、一个句点符号、一个逗号或者是一个行的开始或结尾。假定输入完全由字母、空白字符、逗号和句点构成。输出一行文本中出现的字母数目时，请将字母的大写和小写形式视为同一个字母。按字母顺序输出字母，只列出确实在输入行中出现的字母。例如以下输入行：

```
I say Hi.
```

   它生成的输出如下：

```
3 words
1 a
1 h
2 i
1 s
1 y
```

3. 为具有以下声明的函数给出函数定义。将定义嵌入一个合适的测试程序中。

```
void getDouble(double& inputNumber);
// 后条件：inputNumber 被赋一个由用户核准的值
```

   假定用户采用日常生活中的记数法来输入，比如 23.789，而不是采用科学记数法。定义 getDouble 时，参照图 8.3 给出的 getInt 函数的定义。换言之，函数应该将输入作为字符来读取，编辑字符串，将结果字符串转换成 double 类型的数字。需要定义像 readAndClean 那样的一个函数，而且它比图 8.2 的那个更复杂，因为必须处理小数点。但无论如何，这是一个相当容易的项目。要做一个更困难的项目，请允许用户以日常记数法输入，或者以科学记数法(e 记数法)输入。函数应自行判断输入是否采用了 e 记数法，而不是询问用户是否采用 e 记数法。

4. 写程序读取一个人的姓名，格式如下：首先是名字(FirstName)，然后是中间名或首字母，最后是姓氏(LastName)。然后，程序用以下格式输出姓名：

```
姓氏, 名字 中间名首字母.
```

   例如以下输入：

```
Mary Average User
```

   它生成的输出如下：

```
User, Mary A.
```

换成以下输入：

```
May A. User
```

输出仍然是：

```
User, Mary A.
```

也就是说，必须将中间名转换成首字母缩写形式。即使输入时不包含一个句点，程序也应在中间名首字母后添加句点。程序应允许用户不提供中间名或中间名首字母。在这种情况下，输出中当然也不包含中间名或首字母缩写。例如以下输入：

```
Mary User
```

它生成的输出如下：

```
User, Mary
```

如使用 C 字符串，假定每个姓名的长度不超过 20 个字符。另外，也可在程序中使用 string 类。

提示：输入时可使用三个字符串变量，而不是使用一个较大的字符串变量。做这道题时不用 getline 会容易一些。

5. 写程序读取一行文本，将所有 4 字母单词替换成单词"love"。例如以下输入：

```
I hate you, you dodo!
```

生成的输出应该如下：

```
I love you, you love!
```

当然，输出时并非总能凑出有意义的句子。例如以下输入：

```
John will run home.
```

生成的输出如下：

```
Love love run love.
```

如果 4 字母单词的首字母大写，那么应替换成"Love"，而不是"love"。除了单词首字母，不需要再检查大写。单词被定义为一个由字母构成的字符串，两端可由一个空格、行结束符或者不是字母的其他任何字符来定界。程序应该重复这个操作，直到用户要求退出。

6. 写程序读取一行文本，输出相同的文本，但将所有整数中的数字替换成'x'。
例如，以下输入：

```
My userID is john17 and my 4 digit pin is 1234 which is secret.
```

输出：

```
My userID is john17 and my x digit pin is xxxx which is secret.
```

注意：作为单词一部分的数字不替换成'x'。例如，john17 不能变成 johnxx。使用循环允许重复计算，直到用户要求退出。

7. 写程序为用户输入的句子提供另一个版本，训练用户使用性别中立的语言。程序要求用户输入一个句子，将这个句子读入一个 string 变量，将所有男性代词替换成性别中立的代词。例如，它会将"He"替换成"she or he"。所以，假定输入以下句子：

```
See an adviser, talk to him, and listen to him.
```

应该生成以下建议更换的版本：

```
See an adviser, talk to her or him, and listen to her or him.
```

句子第一个单词首字母必须大写。代词"his"可替换成"her(s)"；你的程序不需要区分"her"和"hers"。允许用户重复输入更多的句子，直到要求结束。

这会是一个很长的程序，需要你极其耐心。如字符串"he"出现在另一个单词中(比如"here")，那么不应该替换。单词定义成由字母构成的字符串，两端可由一个空格、行结束符或者不是字母的其他任何字符来定界。句子最长 100 个字符。

8. 写一个与第 7 章图 7.12 类似的排序函数，但第一个参数是 int 向量而不是数组。该函数不需要图 7.12 那样的 numberUsed 参数，因为向量长度可由成员函数 size()确定。新版排序函数只有一个参数，就是这个 vector 类型的参数。使用选择排序算法(也就是图 7.12 使用的算法)。

9. 重做第 7 章的编程项目 6，用向量取代数组(最好先完成上一个编程项目)。

10. 重做第 7 章的编程项目 5，用向量取代数组。做这个项目前最好先完成项目 8 和项目 9。但需要为这个项目编写自己的(类似的)排序代码，而不是照搬编程项目 8 或项目 9 的排序函数。

11. 假设你的国家发生了战争，敌人用密码系统来通信。现在要对截获到的下面这条消息进行解密：

```
:mmZ\dxZmx]Zpgy
```

消息显然已用敌人的密码系统进行了加密。刚刚获得的情报显示，他们的加密系统基于 ASCII 码。本书附录 3 展示了 ASCII 字符集。字符串中的每个字符都有一个对应的 ASCII 码。例如，字母"A"用数字 65 编码，"B"用数字 66 编码。

敌人的密码系统获取一条消息中的每个字母，并像下面这样加密：

```
If (OriginalChar + Key > 126) then
    EncryptedChar = 32 + ((OriginalChar + Key) -127)
Else
    EncryptedChar = (OriginalChar + Key)
```

例如，假定敌人使用 Key = 10，则消息“Hey”会像下面这样加密：

字符	ASCII 代码
H	72
e	101
y	121

加密的 H = (72 + 10) = 82 = ASCII 中的 R
加密的 e = (101 + 10) = 111 = ASCII 中的 o
加密的 y = 32 + ((121 + 10) - 127) = 36 = ASCII 中的$

最终，"Hey"被加密成"Ro$"来传输。

写程序对截获的消息进行解密。现在只知道敌人使用的 key 是 1～100 的一个数。程序应该使用 1～100 的所有 key 对这条消息进行解密。只有在 key 正确的时候，解密出来的消息才会显得有意义。对于其他所有 key，解密得到的只是垃圾字符。

12. 写程序从控制台输入时间。时间采用“HH:MM AM”或“HH:MM PM”格式输入。允许输入 1 位或 2 位小时数；例如，“1:10 AM”和“11:30 PM”都是允许的。程序包含一个函数，获取包含时间数据的一个 string 参数。函数将时间换算为 24 小时制的 4 位军事时间。例如，“1:10 AM”换算为“0110 hours”，“11:30 PM”换算为“2330 hours”，“12:15 AM”换算为“0015 hours”。自行决定是直接由函数将时间写到控制台，还是返回一个 string，由 main 函数写到控制台。

13. XML(可扩展标记语言)是在 Internet 上构造和存储数据的一种常见格式。下面是一个示例 XML 文件，它在地址簿中存储姓名。在文本编辑器中输入它，保存到名为 address.xml 的文件中[①]。

```
<?xml version="1.0"?>
<address_book>
   <contact>
      <name>George Clooney</name>
      <street>1042 El Camino Real</street>
      <city>Beverly Hills</city>
      <state>CA</state>
      <zip>90214</zip>
   </contact>
```

① 本书配套资源有现成的文件。——译注

```
   <contact>
      <name>Cathy Pearl</name>
      <street>405 A St.</street>
      <city>Palmdale</city>
      <state>CA</state>
      <zip>93352</zip>
   </contact>
   <contact>
      <name>Paris Hilton</name>
      <street>200 S. Elm St.</street>
      <city>Beverly Hills</city>
      <state>CA</state>
      <zip>90212</zip>
   </contact>
   <contact>
      <name>Wendy Jones</name>
      <street>982 Boundary Ave.</street>
      <city>Palmdale</city>
      <state>CA</state>
      <zip>93354</zip>
   </contact>
</address_book>
```

示例文件含有 4 个联系人。<>标记字段开始，而</>标记字段结束。

a. 你准备在加州 Palmdale(state 是 CA，city 是 Palmdale)开一个派对。写程序读取 address.xml 文件，输出住在 Palmdale 的每个人的姓名和地址。程序不应输出任何标记信息，只输出实际数据。
b. 你要向邮编在 90210 到 90214 的每个人发送一份广告传单。写程序读取 address.xml 文件，输出邮编在这个范围的所有人的地址。

可假定地址文件中的每个人都有相同的结构和相同的字段。但是，你的解决方案应该能处理任意数量的联系人，而且不能假定每个联系人的字段都是相同的顺序。

14. 视频讲解：*Solution to Programming Project 8.14*

给定以下函数头：

```
vector<string> split(string target, string delimiter);
```

实现 split 函数，返回由 target 中以 delimiter 分隔的子字符串构成的向量。例如以下代码：

```
split("10,20,30", ",")
```

应返回由字符串"10"，"20"和"30"构成的向量。类似，以下代码：

```
split("do re mi fa so la ti do", " ")
```

应返回由字符串"do"，"re"，"mi"，"fa"，"so"，"la"，"ti"和"do"构成的向量。

15. 写函数判断两个字符串是不是重组字(anagram)。函数对大小写不敏感，并忽略任何标点符号或空格。如果两个字符串的字母能重新排列来构成对方，它们就是重组字。例如，“Eleven plus two”是“Twelve plus one”的重组字。每个字符串都包含 1 个“v”、3 个“e”、2 个“l”等。分别用几个是和不是重组字的字符串测试函数。可以使用 string 类或 C 字符串。

16. 许多赛跑项目要求运动员在鞋子或背心上佩戴 RFID 标签。运动员经过传感器时，电脑记录运动员的号码和当前时间。可沿着赛道布置多个传感器，以精确计算运动员终点时间或配速，并验证运动员经过了所有关键检查点。假定在一场半程马拉松比赛(13.1 英里或 21.1 公里)中使用该系统。只在起点、7 英里和终点部署了传感器。以下是三名运动员的示例数据。第一行是 24 小时制(HH MM SS)的信号枪时间，代表比赛开始。之后的行由传感器记录，内容包括传感器 ID(0 是起点，1 是 7 英里，2 是终点)、运动员号码和时间戳。有的起点时间之所以和信号枪时间不同，是因为运动员的反应速度有快有慢。

```
08 00 00
0,100,08 00 00
0,132,08 00 03
0,182,08 00 15
```

```
1,100,08 50 46
1,182,08 51 15
1,132,08 51 18
2,132,09 34 16
2,100,09 35 10
2,182,09 45 15
```

创建包含示例数据的文本文件。写程序将数据读入数组或向量(可能需要多个)。程序允许用户输入运动员号码，输出该运动员的最终排位、赛道分段配速(单位：分钟/英里，三个传感器将赛道分解成两段)以及总时间和总配速。一个更具挑战性的版本是修改程序来支持沿赛道部署的任意数量的传感器，而非限定三个位置。需提供每个传感器的英里标记。

17. 基于编程项目 16 的数据文件，写程序来检测作弊。符合以下条件即有作弊嫌疑。

    - 运动员错过一个传感器，运动员可能抄了近道。
    - 某段配速过快，运动员可能使用了机动车。快过每英里 4 分 30 秒就有嫌疑。

    输出所有有作弊嫌疑的运动员和理由。

18. 写程序输入两个时间字符串(C 字符串和 STL 字符串都可以)，格式为 HH:MM:SS AM|PM。输出前个时间到后个时间的分秒数。例如，输入以下字符串：

    ```
    11:58:10 PM
    12:02:15 AM
    ```

    程序应输出 4 分 5 秒。

19. 写程序管理最多 10 名玩家的名字及其最高分列表。用两个数组管理。一个存储玩家名字，一个存储最高分。用数组索引同步名字和分数。第 10 章和第 11 章会讲解如何将相关数据放到结构或类中。程序应具有以下功能：

    a. 添加新玩家和分数。如果是 top 10 分数就把它添加到高分列表。同一个人打了多次高分，应允许其多次出现。例如，假定 Bill 玩三次，得分 100，100 和 99；Bob 玩一次，得分 50，则高分列表是 Bill 100, Bill 100, Bill 99, Bob 50。

    b. 打印 top 10 玩家名字和分数，按分数从高到低排列。

    c. 允许用户输入一个玩家的名字，在 top 10 里面就输出其最高分，不在 top 10 里面或者未录入该玩家名字就打印一条消息说明。

    d. 允许用户输入一个玩家的名字，将其最高分从列表中移除。

    创建菜单系统允许用户选择要进行的操作。

第 9 章　指针和动态数组

对一切的理智运用来说，记忆都是必要的。[1]

——*布莱兹·帕斯卡*[2]*(1623—1662)，《思想录》*

概述

指针是能对计算机内存进行更多控制的结构。本章将解释如何将指针用于数组，并介绍一种新的数组类型，即动态数组。动态数组在程序运行时才确定其长度，而不是在写程序时就将长度固定下来。

预备知识

9.1 节讨论指针的基础知识，它基于第 2 章～第 6 章的知识，不需要第 7 章或第 8 章的知识。

9.2 节讨论动态数组，它基于 9.1 节、第 2 章～第 7 章的知识，不需要第 8 章的知识。

9.1　指　　针

不要把指向月亮的手指头当成是月亮。[3]

——*禅语*

指针是变量的内存地址。以前讲过，计算机内存被划分为许多编好号的内存位置(称为字节)，变量被实现为一组相邻的内存位置。还讲过，C++系统有时将内存地址作为变量名使用。假定变量用 3 个内存位置实现，就可以将第一个位置的地址作为变量名使用。例如，假定变量作为传引用参数使用，传给调用函数的就是这个地址，而非变量的标识符名称。

用变量起始处的内存地址命名变量，该地址就称为指针，因为可以认为该地址“指向”变量。之所以说地址“指向”变量，是因为是用变量的位置(而不是变量名)标识变量。例如，在提到位于位置编号 1007 的变量时，可以说：“它是在位置 1007 的变量。”

前面许多地方用过指针。如前所述，在函数调用中将变量作为传引用参数使用时，是以指针形式向函数提供实参。这是指针的一个重要而且强大的用途，但具体过程是由 C++系统自动完成的。本章要解释如何写程序，使其按照自己希望的方式来操纵指针，而不是依赖系统帮助你操纵指针。

---

① 记忆和内存的英文都是 memory。——译注

② 布莱兹·帕斯卡(Blaise Pascal)，法国神学家、基督教哲学家、数学家、物理学家、化学家、音乐家、教育家、气象学家。帕斯卡早期进行自然和应用科学的研究，对机械计算器的制造和流体的研究做出重要贡献，扩展托里切利的工作，澄清了压强和真空的概念。帕斯卡还有力地为科学方法辩护。数学上，帕斯卡促成了两个重要的新研究领域。他是坚定的詹森教派信徒，人文思想大受蒙田影响。宗教论战之作《致外省人书》被奉为法文写作的典范，身后其笔记本被编为《思想录》。

③ 指针就是指针，不是它指向的东西。佛教用“指”喻语言文字，用“月”喻佛法真谛。《楞严经》卷二：“如人以手，指月示人。彼人因指，当应看月。若复观指以为月体，此人岂唯亡失月轮，亦亡其指。何以故？以所标指为明月故。”运用语言宣说佛法，如同用手指指月亮。手指是语言，月亮是真理。痴迷的人，却把手指当成了月亮的本身，岂非大谬不然？！”——译注

指针变量

指针可存储到变量中。但是，虽然指针是内存地址，而且内存地址是数字，但不能将指针存储到 int 或者 double 类型的变量中。容纳指针的变量必须声明为指针类型。例如，以下语句将 p 声明为指针变量，它能容纳指向 double 类型变量的指针：

```
double *p;
```

变量 p 可容纳指向 double 变量的指针，但通常不能容纳指向其他类型(比如 int 或 char)变量的指针。每种变量类型都需要一个不同的指针类型。

通常，为了声明变量来容纳指向特定类型的其他变量的指针，需要像声明那种类型的一个普通变量那样声明指针变量，唯一区别就是在变量名前添加星号。例如，以下语句声明变量 p1 和 p2，使它们能容纳指向 int 类型的变量的指针；另外还声明了 int 类型的两个普通变量，即 v1 和 v2：

```
int *p1, *p2, v1, v2;
```

每个指针变量之前必须有星号。省略上述声明中的第二个星号，p2 就不会成为一个指针变量；相反，它会成为 int 类型的一个普通变量。星号也代表乘法运算，但在现在这种情况下，它具有完全不同的含义。

指针变量的声明

如变量需要容纳指针，而且指针指向 *Type_Name* 类型的其他变量，那么在声明指针变量时，要采用和声明 *Type_Name* 类型的普通变量相似的方式，不同的是必须在变量名之前添加星号。

语法

```
Type_Name *variableName1, *variableName2, …;
```

示例

```
double *pointer1, *pointer2;
```

讨论指针和指针变量时，通常说“指向”，而不是“地址”。假如指针变量 p1 包含变量 v1 的地址，就说指针变量“指向变量 v1”，或者说指针变量是“变量 v1 的指针”。

指针变量(比如前面声明的 p1 和 p2)可包含变量(比如 v1 和 v2)的指针。可用操作符&获取变量地址，再将地址赋给指针变量。例如，以下语句将变量 p1 设为指向变量 v1 的指针：

```
p1 = &v1;
```

现在可采取两种方式称呼 v1，既可直接称为 v1，也可称为“p1 指向的变量”。在 C++中，为了表示“p1 指向的变量”，需要使用*p1。同样的星号在声明 p1 时也用过，但它现在被赋予了全新含义。像这样使用的星号通常称为**提领操作符**，指针变量则认为被**提领**了。[①]

综合运用上述知识，可得到一些令人惊讶的结果。例如以下代码：

```
v1 = 0;
```

① 提领的原文是 dereference，其实就是“取指针指向的值”或“把指针所指的值提领回来”。其他说法包括解引用、逆向引用和用引等。——译注

```
p1 = &v1;
*p1 = 42;
cout << v1 << endl;
cout << *p1 << endl;
```

上述代码的输出如下：

```
42
42
```

由于 p1 包含指向 v1 的指针，所以 v1 和*p1 引用同一个变量。将*p1 设为 42，相当于将 v1 设为 42。

> **地址和数字**
>
> 指针是地址，地址是整数，但指针不是整数。不要吃惊，实情确实如此。C++坚持指针要作为地址使用，不能作为数字使用。指针不是 int 类型或者其他任何数值类型的值。通常不能将指针存储到 int 类型的变量中。如果坚持这样做，大多数 C++编译器都会显示一条错误或警告消息。另外，不能对指针执行标准算术运算(虽然能对指针执行某种形式的加减操作，但这并不是普通的整数加减)。

获取变量地址的符号&也在函数声明中指定传引用参数。这绝非巧合。记住，传引用参数在实现时，是将实参地址传给调用函数。因此，符号&的这两种用法是极其相似的。但由于用法仍然存在少许区别，所以我们将它们视为符号&的不同(但密切相关)的两种用法。

可将指针变量的值赋给另一个指针变量。这会将地址从一个指针变量复制给另一个指针变量。例如，假定 p1 指向 v1，以下语句造成 p2 也指向 v1：

```
p2 = p1;
```

如 v1 的值没有变化，以下语句也会在屏幕上输出 42：

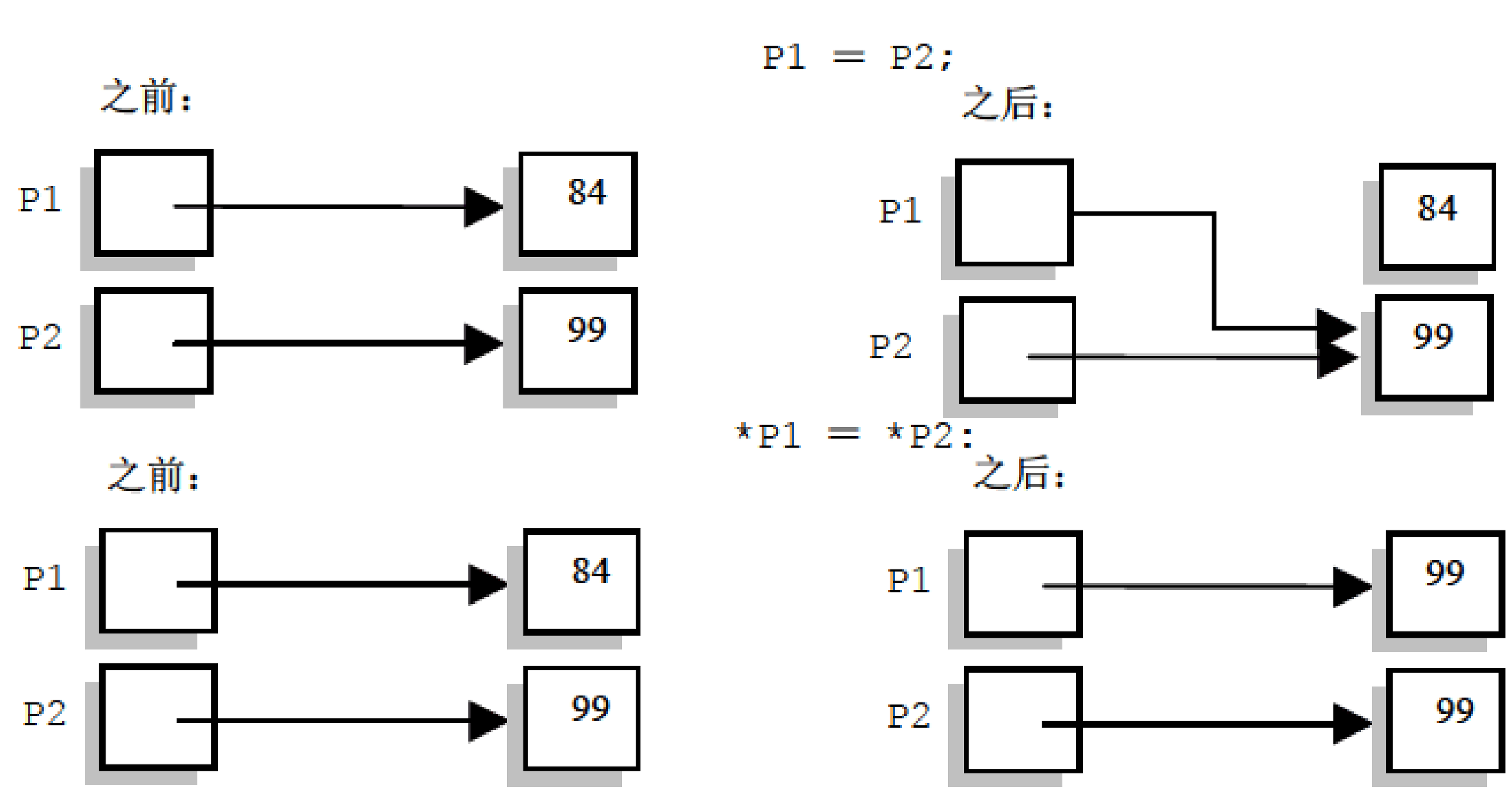
```
cout << *p2;
```

千万不要混淆以下两条语句：

```
p1 = p2;    和    *p1 = *p2;
```

一旦添加了星号，处理的就不再是指针 p1 和 p2，而是由指针指向的变量，如图 9.1 所示。

图 9.1　使用赋值操作符



操作符*和&

指针变量前的操作符*生成它所指向的变量。像这样使用的*操作符称为**提领操作符**。

普通变量前的操作符&生成那个变量的地址；换言之，生成指向变量的指针。操作符&简单地称为**取址(address-of)操作符**。

例如，给定以下声明：

```
double *p, v;
```

以下语句会设置 p 的值，使 p 指向变量 v：

```
p = &v;
```

*p 则生成(提领)p 指向的变量，所以在执行上述赋值语句之后，*p 和 v 引用同一个变量。例如，以下语句将 v 值设为 9.99，即使其中根本没有出现 v 的名称：

```
*p = 9.99;
```

由于指针可以引用变量，所以即使变量没有用标识符命名，也能在程序中操纵该变量。可用操作符 new 创建无标识符的变量。这些无名变量通过指针来引用。例如，以下语句创建 int 类型的一个变量，并将指针变量 p1 设置成这个新变量的地址(换言之，让 p1 指向这个新的无名变量)：

```
p1 = new int;
```

要引用新的无名变量，可以使用*p1(也就是 p1 指向的变量)。处理无名变量时，可采用和处理 int 类型的其他任何变量一样的方式。例如，以下语句从键盘读入一个 int 类型的值，并将其送入无名变量，在它上面加 7，再输出新值：

```
cin >> *p1;
*p1 = *p1 + 7;
cout << *p1;
```

操作符 new 生成新的无名变量，返回指向新变量的指针。要指定新变量的类型，需要在操作符 new 之后添加类型名称。使用操作符 new 创建的变量称为**动态变量**，因其是在程序运行时创建和销毁的。图 9.2 的程序演示了指针和动态变量的一些简单操作。图 9.3 展示了该程序的工作过程。在图 9.3 中，变量用方框表示，变量值写到框内。图中没有显示指针变量实际存储的数值地址。实际数值地址并不重要，重要的是，这些数值代表的是一个特定变量的地址。所以，不是使用实际地址数值，而是只用一个箭头指向具有那个地址的变量。例如，在图 9.3 的(b)部分，p1 包含一个变量的地址，该变量的值未知，所以框中用问号来表示。

图 9.2　基本指针操作

```
1   // 该程序用于演示指针和动态变量
2   #include <iostream>
3   using namespace std;
4
5   int main()
6   {
7       int *p1, *p2;
8
9       p1 = new int;
10      *p1 = 42;
```

```
11      p2 = p1;
12      cout << "*p1 == " << *p1 << endl;
13      cout << "*p2 == " << *p2 << endl;

14      *p2 = 53;
15      cout << "*p1 == " << *p1 << endl;
16      cout << "*p2 == " << *p2 << endl;

17      p1 = new int;
18      *p1 = 88;
19      cout << "*p1 == " << *p1 << endl;
20      cout << "*p2 == " << *p2 << endl;
21      cout << "Hope you got the point of this example!\n";
22      return 0;
23  }
```

示范对话

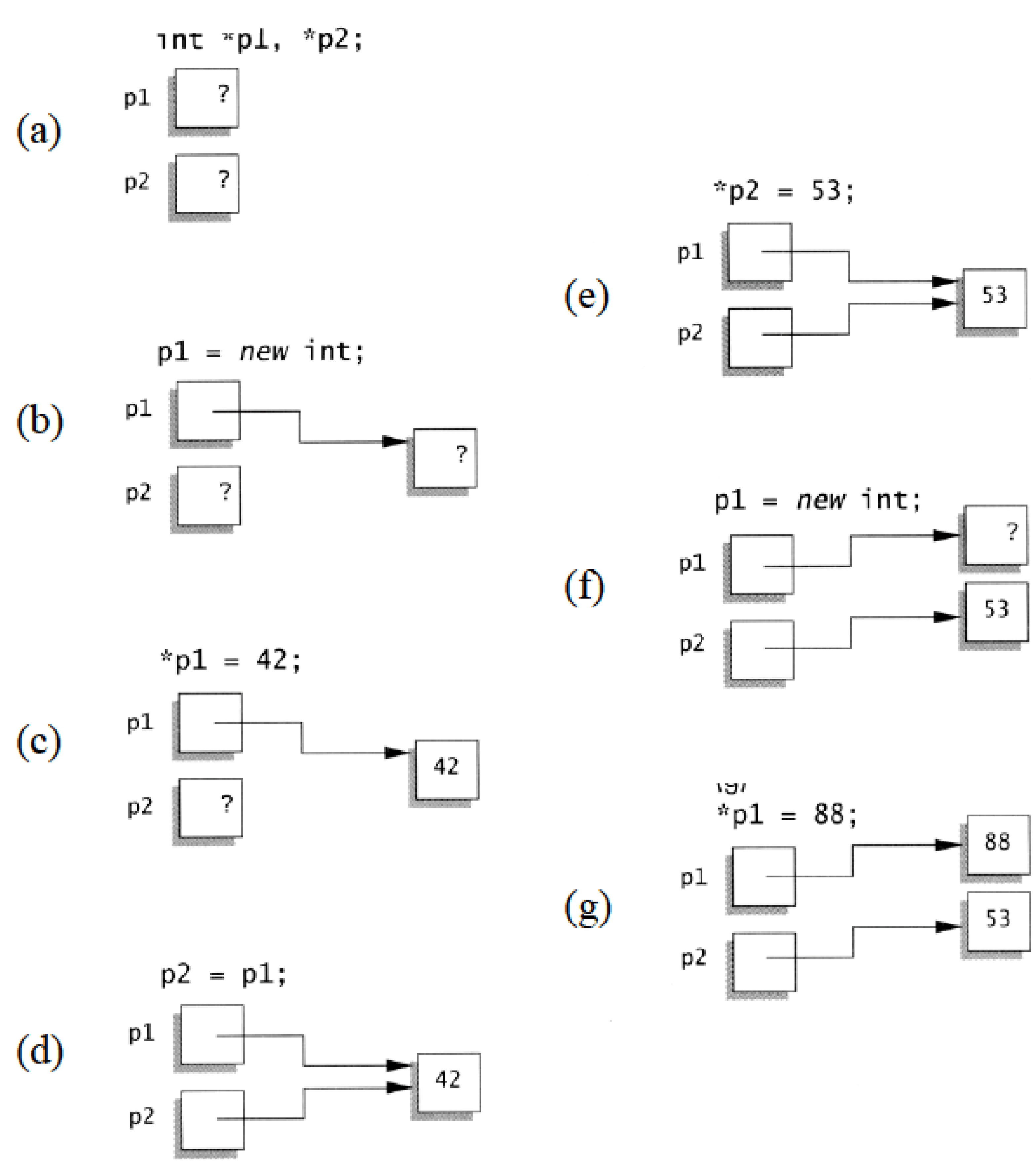
```
*p1 == 42
*p2 == 42
*p1 == 53
*p2 == 53
*p1 == 88
*p2 == 53
Hope you got the point of this example!
```

为指针变量使用操作符=

如果 p1 和 p2 是指针变量，以下语句会修改 p1，使其指向 p2 当前指向的东西：

p1 = p2;

图 9.3　对图 9.2 的解释



操作符 new

操作符 new 创建指定类型的一个新动态变量，返回指向新变量的指针。例如，以下语句新建一个 MyType 类型的动态变量，让指针变量 p 指向这个新变量：

```
MyType *p
p = new MyType;
```

如果类型是带有构造函数的类，则会为新建的动态变量调用默认构造函数。也可以指定初始化列表，从而调用其他构造函数版本：

```
int *n;
n = new int(17);  // 将 n 初始化成 17
MyType *mtPtr;
mtPtr = new MyType(32.0, 17);  // 调用 MyType(double, int);
```

C++标准规定，若没有足够内存创建新变量，操作符 new 默认会终止程序。①

√ 自测题

1. 简单解释什么是 C++指针。
2. 以下声明可能让人产生什么误解？

```
int* intPtr1, intPtr2;
```

3. 用实际的例子至少给出操作符*的两种用法。指出*所做的事情，并为*的每种用法命名。
4. 以下代码输出什么？

```
int *p1, *p2;
p1 = new int;
p2 = new int;
*p1 = 10;
*p2 = 20;
cout << *p1 << " " << *p2 << endl;
p1 = p2;
cout << *p1 << " " << *p2 << endl;
*p1 = 30;
cout << *p1 << " " << *p2 << endl;
```

对于以下语句：

```
*p1 = 30;
```

如果把它替换成以下语句：

```
*p2 = 30;
```

输出有什么改变？

5. 以下代码输出什么？

```
int *p1, *p2;
p1 = new int;
p2 = new int;
*p1 = 10;
*p2 = 20;
cout << *p1 << " " << *p2 << endl;
*p1 = *p2; // 这里有别于自测题 4
cout << *p1 << " " << *p2 << endl;
```

① 从技术上说，操作符 new 会抛出一个异常；如果没有捕捉这个异常，就会终止程序。完全可以“捕捉”异常，或者准备一个 new 处理程序，这些主题将在第 16 章讨论。

```
*p1 = 30;
cout << *p1 << " " << *p2 << endl;
```

基本内存管理

系统为动态变量保留一个特殊内存区域，称为**自由存储**[①]。程序新建的任何动态变量都会消耗自由存储中的部分内存。程序创建太多动态变量可能耗尽自由存储。这时任何更多的 new 调用都会失败。

在不同 C++实现中，自由存储大小也不同。通常很大，一般的程序不太可能耗尽。但作为良好编程习惯，即使最普通的程序，也应回收任何不再需要的自由存储内存。如程序不再需要一个动态变量，就主动回收其使用的内存。操作符 delete 可销毁动态变量，并将其占用的内存还给自由存储，使内存得以重用。假定 p 是指向动态变量的指针变量，以下语句会销毁 p 指向的动态变量，将占用的内存还给自由存储：

```
delete p;
```

像这样调用 delete，p 的值会进入未定义状态，现在的 p 应被视为一个未初始化变量。

> **操作符 delete**
>
> 操作符 delete 销毁动态变量，将其占用的内存还给自由存储。这样就可重用那些内存来创建新的动态变量。例如，以下语句销毁指针变量 p 指向的动态变量：
>
> ```
> delete p;
> ```
>
> 像这样调用了 delete，指针变量(比如 p)的值会变成未定义(如动态变量是数组，要用一个稍微不同的 delete 版本，详情将在本章后面讨论)。

陷阱：虚悬指针

为指针变量使用 delete 后，它指向的动态变量会被销毁。之后，指针变量的值进入未定义状态；也就是说，你不知道它指向哪里，也不知道它指向的地方有什么值。此外，如果另一个指针变量也指向被销毁的动态变量，那么那个指针变量也会进入未定义状态。这些未定义的指针变量称为**虚悬指针**。假定 p 是虚悬指针，而程序将提领操作符*应用于 p(表达式*p)，将产生不可预料的(而且通常都是灾难性的)后果。将提领操作符*应用于一个指针变量之前，先确定指针变量指向一个实际存在的变量。 ■

静态变量和自动变量

操作符 new 创建的变量称为动态变量，因为它们在程序运行时创建和销毁。与动态变量相比，普通变量似乎是静态的。但 C++程序员使用的术语比这复杂，普通变量不能称为静态变量。

---

① freestore，有时也称为“堆”(heap)。但是，两者并不是完全没有区别。堆和自由存储可能存在于不同的物理内存区域，由不同的底层内存管理器进行控制。从技术上说，自由存储是一个抽象的术语，指可供动态分配的未使用内存。而堆是一个数据模型，几乎所有 C++编译器都用它实现自由存储。现实中，堆和自由存储的区别大致就是 C 和 C++的内存模型的区别。——译注

我们以前使用的普通变量实际并不是真正静态的。如变量是函数的局部变量，调用函数时变量会由 C++系统创建，并在函数调用结束时销毁。由于程序的 main 部分本质上是一个名为 main 的函数，所以在 main 部分中声明的变量也会这样处理。由于对 main 的调用直到程序终止才结束，所以 main 中声明的变量直到程序终止才销毁。无论 main 还是其他函数，局部变量都这样处理。我们过去用过的普通变量(也就是在 main 或者其他函数定义中声明的变量)称为**自动变量**，因为它们的动态属性由系统自动控制；调用函数时，函数内部声明的变量自动创建，函数调用结束后，自动销毁。我们通常将这些变量称为**普通变量**，但也有一些书把它们称为自动变量。

还有一种变量称为**全局变量**。全局变量是在任何函数定义(包括 main)的外部声明的变量。我们在第 4 章简单讨论了全局变量。正如当时指出的那样，我们不需要全局变量，也没有用它们。

编程提示：定义指针类型

可定义一个指针类型名称，使指针变量能像其他变量那样声明，而不需要在每个指针变量前面添加星号。例如，以下语句定义了名为 IntPtr 的指针变量类型，变量中包含指向 int 变量的指针：

```
typedef int* IntPtr;
```

所以，以下两个指针变量声明等价：

```
IntPtr p;
int *p;
```

可用 typedef 为“任何类型名称或定义”定义别名。例如，以下语句定义名为 Kilometers 的类型，它等同于类型名称 double：

```
typedef double Kilometers;
```

给定以上类型定义，就可以像下面这样定义 double 类型的变量：

```
Kilometers distance;
```

重命名现有类型偶尔有一些用处，但 typedef 主要用途还是为指针变量定义类型。

使用已定义的指针类型名称(比如上面定义的 IntPtr)有两方面的好处。首先，它避免了不小心遗漏星号。记住，如本意是让 p1 和 p2 成为指针，以下声明不能满足你的愿望：

```
int *p1, p2;
```

由于 p2 遗漏了*，所以变量 p2 只是普通 int 变量，而非指针变量。如一时头脑发热将*附加到 int 后面，那么问题一样，只是更难发现。C++允许为类型名称(比如 int)附加星号，所以以下语句合法：

```
int* p1, p2;
```

合法但易误导人。表面上 p1 和 p2 都是指针变量，但实际只有 p1 才是；p2 是普通 int 变量。就 C++编译器而言，附加到标识符 int 后面的星号就是附加到标识符 p1 前面的。将

p1 和 p2 都声明为指针变量的正确方式是：

```
int *p1, *p2;
```

但要将 p1 和 p2 都声明为指针变量，一种更简单、更不容易出错的方式就是使用已定义类型名称 IntPtr。如下所示：

```
IntPtr p1, p2;
```

定义函数时，将指针变量作为传引用参数，将体现出已定义指针类型(比如 IntPtr)的第二个好处。不利用已定义的指针类型名称，就需要在函数声明中同时用到符号*和&，这样容易使人困惑。为指针类型使用类型名称后，指针类型的传引用参数就没这么复杂了。指定已定义指针类型的一个传引用参数时，和指定其他任何传引用参数是相似的。例如：

```
void sample_function(IntPtr& pointer_variable);
```

■

类 型 定 义

可以为类型定义分配名称，再用这个名称声明变量。这是用关键字 typedef 来实现的。这些类型定义通常放到程序 main 主体 (以及其他函数体)的外部。可用这个办法为指针类型指定别名，如下例所示。

语法

```
typedef Known_Type_Definition New_Type_Name;
```

示例

```
typedef int* IntPtr;
```

以后就可用类型名称 IntPtr 来声明指向 int 类型的动态变量的指针，如下所示：

```
IntPtr pointer1, pointer2;
```

√ 自测题

6. 假定像下面这样创建动态变量：

```
char *p;
p = new char;
```

   假定指针变量 p 的值没有改变(仍然指向相同的动态变量)，如何销毁这个新的动态变量，并将它占用的内存还给自由存储，以便重用这些内存来创建新的动态变量？

7. 为名为 NumberPtr 的类型写一个定义，它是指针变量类型，容纳的指针指向 int 类型的动态变量。再为名为 myPoint 的指针变量写一个声明，该指针变量的类型是 NumberPtr。

8. 描述操作符 new 的行为，操作符 new 会返回什么？

9.2　动 态 数 组

通过本节的学习，你将知道数组变量实际是指针变量。另外，还将体验如何用动态数组写程序。**动态数组**是写程序时不指定长度的数组，它的长度在程序运行时确定。

数组变量和指针变量

第 7 章讲解了数组在内存中如何存储。那时还没有学习指针，所以是利用内存地址的概念讨论数组。但是，内存地址是指针。所以，C++数组变量实际是指针变量，指向数组的第一个索引变量。给定以下两个变量声明，p 和 a 是同类型的变量：

```
int a[10];
typedef int* IntPtr;
IntPtr p;
```

图 9.4 证明 a 和 p 是同类型的变量。

图 9.4　数组和指针变量

```
1  // 该程序用于证明数组变量其实就是指针变量
2  #include <iostream>
3  using namespace std;
4
5  typedef int* IntPtr;
6
7  int main()
8  {
9      IntPtr p;
10     int a[10];
11     int index;
12
13     for (index = 0; index < 10; index++)
14         a[index] = index;
15
16     p = a;
17
18     for (index = 0; index < 10; index++)
19         cout << p[index] << " ";
20     cout << endl;
21
22     for (index = 0; index < 10; index++)
23         p[index] = p[index] + 1;
24
25     for (index = 0; index < 10; index++)
26         cout << a[index] << " ";
27     cout << endl;
28
29     return 0;
30 }
```

注意，修改数组 p 会造成对数组 a 进行同样的修改

输出

```
0 1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9 10
```

由于 a 是指向 int 变量(也就是 a[0]，数组第一个元素)的指针，所以它的值可以像下面这样赋给指针变量 p：

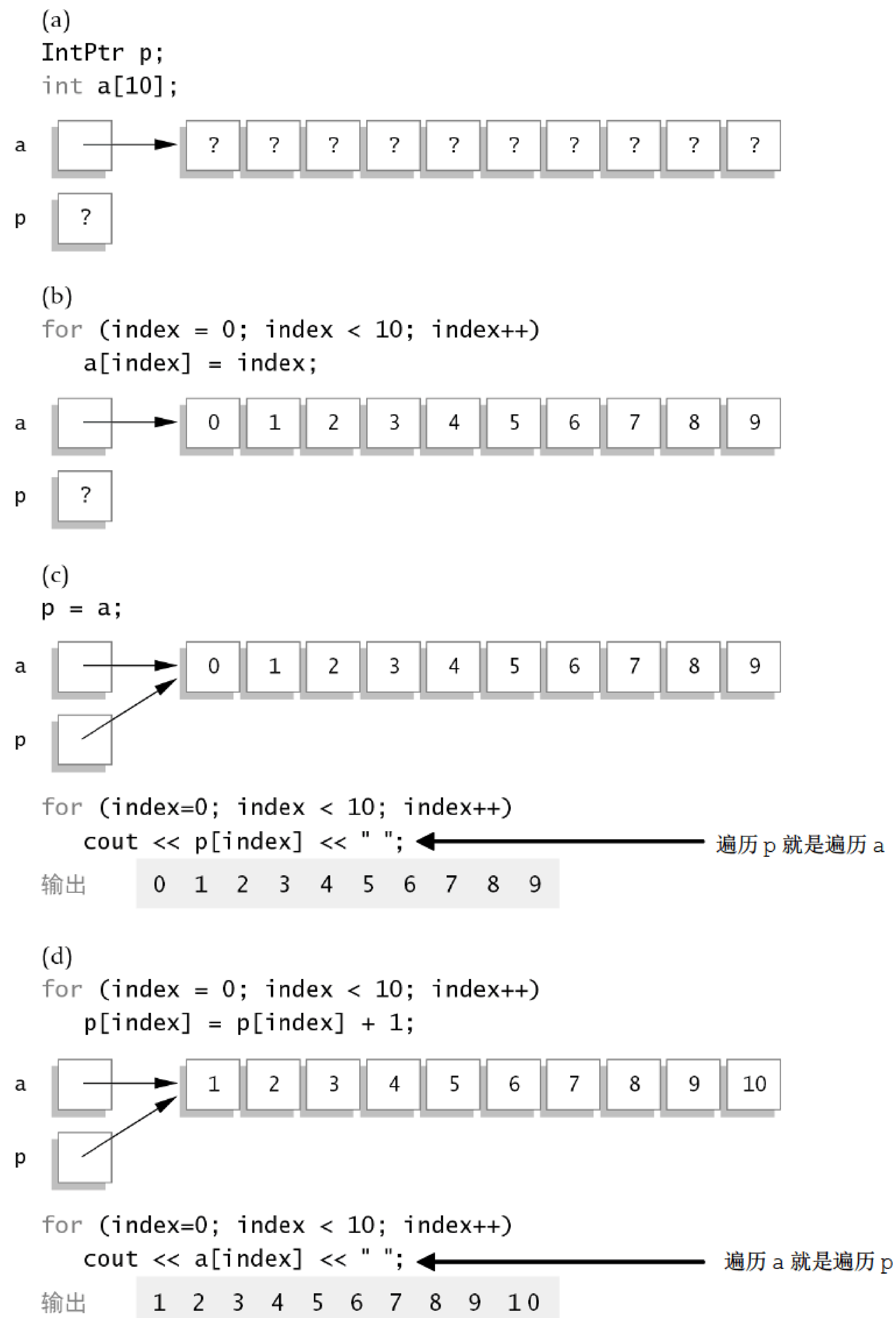
```
p = a;
```

像这样赋值后，p 和 a 便都指向同一个内存位置。所以，p[0]，p[1]，…，p[9]引用的就是索引变量 a[0]，a[1]，…，a[9]。只要指针变量指向内存中的数组，以前适于数组的方括号记号法就同样适于指针变量。经过上述赋值后，可将标识符 p 视为数组标识符。还可将标识符 a 视为指针变量。但有一个重要区别，不可更改数组变量(比如 a)中的指针值。例如，以下语句是非法的：

```
IntPtr p2;
.. // 为p2 赋一个指针值
a = p2; // 非法，不能为 a 赋不同的地址
```

图 9.5 展示了图 9.4 的程序的工作原理。和图 9.3 一样，变量用方框表示，变量值写到框内。箭头代表指向另一个内存位置的指针或引用。本例是指向数组第一个元素。

图 9.5　对图 9.4 的解释



创建和使用动态数组

以前使用数组时必须指定长度。但有的时候，除非程序实际运行，否则不好确定长度。例如，数组可能容纳了一个学生 ID 列表，但程序每次运行时，班级的学生数都可能发生变化。沿用传统做法，就必须预估数组可能的最大长度，并希望那个长度足够大，能适应所

有情况。但这样做有两个问题。首先，你估计的长度可能还是太小，造成程序不能适应所有情况。其次，由于数组可能包含许多未使用的位置，所以会浪费计算机内存。动态数组避免了所有这些问题。用动态数组容纳学生 ID，可在程序运行时输入班级的学生数。然后，程序会创建刚好那么大的动态数组。

动态数组用操作符 new 创建。和许多人想象的不同，动态数组的创建和使用其实非常简单。由于数组变量其实就是指针变量，所以可以用操作符 new 创建被用作数组的动态变量，并像使用普通数组那样使用动态数组变量。例如，以下语句创建一个动态数组变量，其中含有 10 个 double 类型的数组元素：

```
typedef double* DoublePtr;
DoublePtr p;
p = new double[10];
```

要在动态数组中包含其他任何类型的元素，只需将 double 替换成所需的类型。要获得其他长度的动态数组变量，只需将 10 替换成所需的长度。

这个例子有几个或明显或不明显的问题要注意。首先，为动态数组指针使用的指针类型和为单个数组元素使用的指针类型是相同的。例如，对于由 double 类型的元素构成的数组，它的指针类型就是为 double 类型的普通变量使用的指针类型。指向数组的指针实际是指向数组第一个索引变量(数组元素)的指针。上例会创建含有 10 个索引变量的一个完整数组。然后，指针 p 指向这 10 个索引变量的第一个。

其次，调用 new 时，动态数组长度在类型名称(本例是 double)之后的方括号中给出。这就告诉计算机要为动态数组保留多大存储空间。如省略方括号和其中的 10，计算机就只为 double 类型的一个变量分配足够大的存储空间，而不是分配 10 个 double 类型的索引变量所需的空间。如图 9.6 所示，可用 int 变量代替常量 10，到程序运行时才读入动态数组的长度。

图 9.6　动态数组

```
1   // 对通过键盘输入的一组数字进行排序
2   #include <iostream>
3   #include <cstdlib>
4   #include <cstddef>
5
6   typedef int* IntArrayPtr;
7
8   void fill_array(int a[], int size);
9   // 前条件：size 是数组 a 的长度
10  // 后条件：a[0]～a[size-1]使用从键盘输入的值来填充
11
12  void sort(int a[], int size);
13  // 前条件：size 是数组 a 的长度
14  // 数组元素 a[0]～a[size-1]已被赋值
15  // 后条件：重新排列 a[0]～a[size-1]的值，使 a[0] <= a[1] <= ... <= a[size-1]
16
17  int main()
18  {
19      using namespace std;
20      cout << "This program sorts numbers from lowest to highest.\n";
21
22      int arraySize;
23      cout << "How many numbers will be sorted? ";
24      cin >> arraySize;
25
26      IntArrayPtr a;
27      a = new int[arraySize];
```

（第 8 行和第 12 行旁注：普通数组参数）

```
28
29      fill_array(a, arraySize);
30      sort(a, arraySize);
31
32      cout << "In sorted order the numbers are:\n";
33      for (int index = 0; index < arraySize; index++)
34          cout << a[index] << " ";   ← 像使用普通数组那样
35      cout << endl;                     使用动态数组 a
36
37      delete [] a;
38
39      return 0;
40  }
41
42  // 使用 iostream 库:
43  void fill_array(int a[], int size)
44  {
45      using namespace std;
46      cout << "Enter " << size << " integers.\n";
47      for (int index = 0; index < size; index++)
48          cin >> a[index];
49  }
50
51  void sort(int a[], int size)
```

<可使用 sort 的任何实现。可能需要、也可能不需要一些附加的函数定义。在你的实现中，甚至不需要知道是否在为动态数组调用 sort。例如，可选择图 7.12 的实现(只需适当调整参数名)>

图 9.6 的程序对一个数字列表进行排序。程序支持任意长度的数字列表，因其使用动态数组来容纳数字。数组长度在程序运行时确定。程序首先询问用户有多少个数，然后用操作符 new 创建刚好那么大的动态数组。动态数组长度由变量 arraySize 给定。

请注意 delete 语句，它在图 9.6 中销毁动态变量 a。由于程序马上就要终止，所以并不是真的需要这个 delete 语句。但如果程序准备用动态变量做其他事情，就应该包括这个 delete 语句，将动态数组占用的内存还给自由存储。为动态数组执行的 delete 语句类似于以前介绍过的 delete 语句，只是在动态数组的情况下，必须包括一对空的方括号，如下所示：

```
delete [] a;
```

方括号告诉 C++要销毁的是动态数组变量，所以系统会检查数组的长度，并删除刚好那么多的索引变量。如省略方括号，相当于告诉计算机只销毁一个 int 类型的变量。例如：

```
delete a;
```

虽然上述语句不合法，但大多数编译器都检测不到这个错误。ANSI C++标准规定，这种情况下发生的事情是“未定义”的，意味着编译器的作者可以做他觉得方便的任何事情——注意只是编译器的作者方便，而不是我们方便。即使会做一些有用的事情，也不能保证编译器的下个版本或其他任何编译器做同样的事情。总之，使用下面这样的语句分配了内存之后：

```
arrayPtr = new MyType[37];
```

务必使用以下语句回收内存：

```
delete [] arrayPtr;
```

创建动态数组时，要调用 new 创建指针变量(比如图 9.6 的指针 a)。调用 new 之后，不能再把其他任何指针值赋给这个指针变量，否则以后调用 delete 将动态数组占用的内存

还给自由存储时会发生混乱。

动态数组用 new 和一个指针变量创建。程序结束使用动态数组后，调用 delete[]将数组占用的内存还给自由存储。除此之外，动态数组的用法和其他数组没有任何区别。

怎样使用动态数组

- **定义指针类型**　定义的指针类型就是为单个数组元素使用的指针类型。例如，对于 double 类型的动态数组，可像下面这样定义指针类型：

  ```
  typedef double* DoubleArrayPtr;
  ```

- **声明指针变量**　声明该定义类型的一个指针变量。指针变量指向内存中的动态数组，并被用作动态数组名称：

  ```
  DoubleArrayPtr a;
  ```

- **调用 new**　使用操作符 new 创建一个动态数组：

  ```
  a = new double[arraySize];
  ```

  动态数组长度在方括号内给出。可用 int 变量或其他 int 表达式给出长度。在上例中，arraySize 可以是值在运行才确定的 int 变量。

- **和普通数组一样使用**　指针变量(比如 a)可以像普通数组那样使用。例如，可采用标准方式书写索引变量，比如 a[0]，a[1]，等等。不能再为指针变量赋其他任何指针值。相反，它应该像数组变量那样使用。

- **调用 delete []**　用完动态数组后使用 delete、一对空的方括号和指针变量来销毁动态数组，将其占用的内存还给自由存储以便重用。例如：

  ```
  delete [] a;
  ```

√ 自测题

9. 为指向动态数组的指针变量写类型定义。数组元素类型是 char，类型名为 CharArray。

10. 假定程序包含用于创建动态数组的代码，如下所示：

    ```
    int *entry;
    entry = new int[10];
    ```

    所以，指针变量 entry 指向这个动态数组。请编写代码，用从键盘输入的 10 个数字填充数组。

11. 假定程序包含自测题 10 的用于创建动态数组的代码，并假定指针变量 entry 没有改变它的(指针)值。请编写代码销毁这个新的动态数组，将占用的内存还给自由存储。

12. 以下代码输出什么？(假定已嵌入一个正确且完整的程序中。)

    ```
    int a[10];
    int *p = a;
    int i;
    for (i = 0; i < 10; i++)
        a[i] = i;
    for (i = 0; i < 10; i++)
       cout << p[i] << " ";
    cout << endl;
    ```

13. 以下代码输出什么？(假定已经嵌入一个正确且完整的程序中。)

```
int arraySize = 10;
int *a;
a = new int[arraySize];
int *p = a;
int i;
for (i = 0; i < arraySize; i++)
    a[i] = i;
p[0] = 10;
for ( i = 0; i < arraySize; i++)
    cout << a[i] << " ";
cout << endl;
```

指针运算(选读)

视频讲解：*Dynamic Arrays and Pointer Arithmetic*

指针运算是地址运算而非数字运算。例如，假定程序包含以下代码：

```
typedef double* DoublePtr;
DoublePtr d;
d = new double[10];
```

执行这些语句后，d 包含索引变量 d[0]的地址。表达式 d + 1 的求值结果是 d[1]的地址，d + 2 是 d[2]的地址，依此类推。注意，虽然 d 是地址，而地址是数字，但 d + 1 不是简单地为 d 中的数字加上 1。如果 double 类型的变量要占用 8 个字节(8 个内存位置)，而且 d 包含的地址是 2001，那么 d + 1 的求值结果是内存地址 2009。当然，可将 double 类型替换为其他任何类型。无论如何，指针加法的递增单位就是那个类型的一个变量占用的字节数。

这种指针运算提供了对数组进行操纵的另一种方式。例如，假定 arraySize 是 d 指向的动态数组的长度，那么以下语句输出动态数组的内容：

```
for (int i = 0; i < arraySize; i++)
    cout << *(d + i) << " ";
```

等价于：

```
for (int i = 0; i < arraySize; i++)
    cout << d[i] << " ";
```

不能为指针执行乘法或除法运算。唯一能做的就是在指针上加一个整数，从指针上减一个整数，或者让相同类型的两个指针相减。对两个指针执行减法运算，结果是两个地址之间的索引变量的数目。记住，要求两个指针值的差值，这两个值必须指向同一个数组！如一个指针指向一个数组，另一个指向另一个，求这两个指针的差值是没有意义的。可以使用递增和递减操作符++和--。例如，d++使 d 的值递增，使其包含下一个索引变量的地址。d--使 d 的值递减，使其包含上一个索引变量的地址。

√ 自测题

(以下两题对应于 9.2 节所介绍的内容)

14. 以下代码输出什么？(假定已嵌入一个正确且完整的程序中。)

```
int arraySize = 10;
int *a;
```

```
a = new int[arraySize];
int i;
for (i = 0; i < arraySize; i++)
    *(a + i) = i;
for (i = 0; i < arraySize; i++)
     cout << a[i] << " ";
cout << endl;
```

15. 以下代码输出什么？(假定已嵌入一个正确且完整的程序中。)

```
int arraySize = 10;
int *a;
a = new int[arraySize];
int i;
for (i = 0; i < arraySize; i++)
     a[i] = i;
while (*a < 9)
{
    a++;
    cout << *a << " ";
}
cout << endl;
```

多维动态数组(选读)

可以使用多维动态数组，但记住多维数组实际是数组构成的数组，每个子数组还可能由其他子数组构成。例如，要创建二维动态数组，必须理解它是由数组构成的一个数组。要创建二维整数数组，先要创建一个一维动态数组，它由 int*类型的指针构成(int*是一维 int 数组的类型)。然后，要为这个指针数组的每个索引变量都创建一个 int 动态数组。

通过分析具体的类型定义，可以更容易理解这一点。下面这行代码为一个普通的一维 int 动态数组定义了变量类型：

```
typedef int* IntArrayPtr;
```

为了获得 3×4 的 int 数组，需要基类型为 IntArrayPtr 的数组。例如：

```
IntArrayPtr *m = new IntArrayPtr[3];
```

这就创建了一个由 3 个指针构成的数组，每个指针都能命名一个 int 动态数组。例如：

```
for (int i = 0; i < 3; i++)
    m[i] = new int[4];
```

最终的数组 m 就是一个 3×4 的动态数组。图 9.7 用一个简单的程序进行了演示。

图 9.7　一个二维动态数组

```
1   #include <iostream>
2   using namespace std;
3
4   typedef int* IntArrayPtr;
5
6   int main()
7   {
8       int d1, d2;
9       cout << "Enter the row and column dimensions of the array:\n";
10      cin >> d1 >> d2;
11
12      IntArrayPtr *m = new IntArrayPtr[d1];
13      int i, j;
14      for (i = 0; i < d1; i++)
15          m[i] = new int[d2];
16      // m 现在成为一个 d1×d2 的数组
17
```

```
18      cout << "Enter " << d1 << " rows of "
19           << d2 << " integers each:\n";
20      for (i = 0; i < d1; i++)
21          for (j = 0; j < d2; j++)
22              cin >> m[i][j];
23
24      cout << "Echoing the 2 dimensional array:\n";
25      for (i = 0; i < d1; i++)
26      {
27          for (j = 0; j < d2; j++)
28              cout << m[i][j] << " ";
29          cout << endl;
30      }
31      for (i = 0; i < d1; i++)
32          delete[] m[i];
33      delete[] m;
34
35      return 0;
36  }
```

注意，对于创建数组的每个 new 函数调用，都必须有一个对应的 delete[] 调用(由于本程序马上就要结束，所以这里实际并不需要包含它们，但在其他情况下，务必将它们包含在内，这很重要)

示范对话

```
Enter the row and column dimensions of the array:
3 4
Enter 3 rows of 4 integers each:
1 2 3 4
5 6 7 8
9 0 1 2
Echoing the two-dimensional array:
1 2 3 4
5 6 7 8
9 0 1 2
```

注意 delete 在图 9.7 中的运用。由于 m 是数组构成的数组，所以必须为 for 循环中用 new 创建的每个数组调用 delete[]，以便释放并回收它们占用的内存。最后，还要再调用一次 delete[]，对数组 m 本身进行回收。创建数组的每个 new 调用都必须有一个对应的 delete []调用。由于示范程序在调用了 delete[]之后终止，所以本例可以省略这些调用。但有必要理解它们的用法，并养成使用它们的良好习惯。

小　结

- 指针是内存地址，所以通过对变量在计算机内存中的地址进行命名，指针提供了一种间接的变量命名方式。
- 动态变量是程序运行时创建(和销毁)的变量。
- 动态变量要占用计算机内存的一个特殊区域，这个区域称为自由存储。程序结束动态变量的使用后，应该将动态变量占用的内存还给自由存储，以便重用；这是用 `delete` 语句来完成的。
- 动态数组是其长度在程序运行时确定的数组。动态数组被实现为数组类型的动态变量。

自测题答案

1. 指针是变量的内存地址。

2. 粗心大意的人和新手一看到 int*，可能会认为它声明了指针类型的两个对象，这些指针指向 int 变量。遗憾的是，符号*绑定到标识符而不是类型(换言之，不和 int 绑定)。这个语句的结果是将 intPtr1 声明为 int 指针，intPtr2 还是一个普通 int 变量。

3.
```
int *p;      // 声明指针变量，容纳指向 int 变量的指针
*p = 17      // 这里的符号*是提领操作符，将 17 赋给 p 指向的内存位置
```

4.
```
10 20
20 20
30 30
```
将*p1 = 30;替换成*p2 = 30;，输出结果是一样的。

5.
```
10 20
20 20
30 20
```

6.
```
delete p;
```

7.
```
typedef int* NumberPtr;
NumberPtr myPoint;
```

8. new 操作符获取一个类型作为参数，它在自由存储中为那种类型的一个变量分配恰当的空间。如果自由存储有足够的可用内存，就返回指向那个内存空间的指针(也就是指向新的动态变量的指针)。内存不够，程序终止。

9.
```
typedef char* CharArray;
```

10.
```
cout << "Enter 10 integers:\n";
for (int i = 0; i < 10; i++)
   cin >> entry[i];
```

11.
```
delete [] entry;
```

12. `0 1 2 3 4 5 6 7 8 9`

13. `10 1 2 3 4 5 6 7 8 9`

14. `0 1 2 3 4 5 6 7 8 9`

15. `1 2 3 4 5 6 7 8 9`

编 程 练 习

编程练习一般只需写很小的程序，运用本章提到的编程概念。

1. C 语言没有提供传引用语法将变量以传引用的方式传给函数。相反，变量以传指针的方式传递(更让人迷惑的是，有时会将传指针称为传引用)。本练习要求你做和 C 一样的事情。事实上，使用 C++的引用参数语法来实现这个机制会更简单一些。以下函数获取一个整数指针：
```
void addOne(int *ptrNum)
```
完成函数定义，在 ptrNum 引用的整数上加 1。写 main 函数定义整数变量，赋予初始值，调用 addOne 并输出变量。结果应该加 1。

2. 写程序要求用户输入整数并赋给 numDoubles 变量。创建动态数组来存储该变量的值，用循环允许用户将 double 值输入每个数组元素。遍历数组，计算并输出平均值。退出前删除分配给动态数组的内存。

3. 该练习要求阅读关于指针运算的选读内容，然后完成 isPalindrome 函数，字符串 cstr 是回文(顺读倒读都一样)就返回 true，否则返回 false。函数使用 cstring 库。

```
bool isPalindrome(char* cstr)
{
   char* front = cstr;
   char* back = cstr + strlen(cstr)-1;

   while (front < back)
   {
      // 完成这里的代码
   }
   return true;
}
```

以下示例 main 函数用于测试：

```
int main()
{
   char s1[50] = "neveroddoreven";
   char s2[50] = "not a palindrome";
   cout << isPalindrome(s1) << endl; // true
   cout << isPalindrome(s2) << endl; // false
   return 0;
}
```

编 程 项 目

编程项目要求综合运用多方面的知识来解决问题，程序一般比编程练习大，解题方式多样化。

1. 用动态数组重做第 7 章的编程项目 7。在这个版本中，用动态数组来存储每个大整数中的数位。要允许任意数量的数位，而不要将最大位数限定为 20。

2. 重做第 7 章的编程项目 3。在这个版本中，要返回一个新的动态数组，其中已经删除了所有重复的字母，而不是对部分填充的数组进行修改。数据不需要的时候，不要忘记释放为返回的动态数组分配的内存。

3. 使用一个或多个动态数组重做第 7 章的编程项目 11。在这个版本中，程序询问用户一架飞机有多少排(行)座位，并能处理那么多排(行)的座位，而不是像第 7 章的编程项目 11 那样，假定飞机只有 7 排(行)座位。

4. 写函数获取一个 C 字符串作为参数，并反转字符串。函数应使用两个指针：front 和 rear。front 指针最初引用字符串中的第一个字符，rear 指针最初引用字符串中的最后一个字符。为了反转这个字符串，需要反转由 front 和 rear 引用的字符，然后递增 front，让它指向下一个字符；并递减 rear，让它指向前一个字符。依此类推，直到整个字符串都得到反转。写一个 main 程序来测试函数，要求测试字符串既有偶数长度的，也有奇数长度的。

5. 假定你主管 4 个计算机实验室。每个实验室安置的计算机都按下表进行编号。

实验室编号	计算机编号
1	1—5
2	1—6
3	1—4
4	1—3

每个用户都有唯一的 5 位 ID。用户在任何时候登录，他的用户 ID、实验室编号以及计算机编号都会传输到你的系统上。例如，假定用户 49193 登录到实验室 3 的工作站 2 上，那么你的系统会接收到

(49193,2,3)作为输入。类似地，当用户从一台计算机上注销时，系统也会接收到实验室编号和计算机编号。

写程序跟踪每个实验室每个用户登录每台计算机的情况。例如，假定用户 49193 登录到实验室 3 的计算机 2 上，而用户 99577 登录到实验室 4 的计算机 1 上，你的系统会显示以下信息：

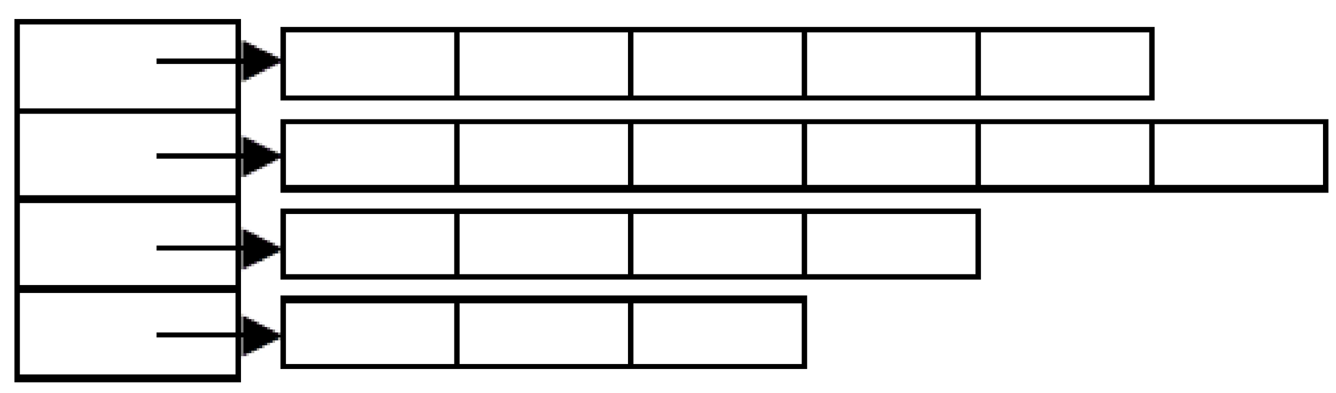
```
Lab Number Computer Stations
1          1: empty    2: empty    3: empty 4: empty 5: empty
2          1: empty    2: empty    3: empty 4: empty 5: empty 6: empty
3          1: empty    2: 49193    3: empty 4: empty
4          1: e99577   2: empty    3: empty
```

创建一个菜单，允许管理员手动输入登录或注销数据来模拟信息传输。任何时候有人登录或注销，屏幕显示都应当更新。同时写一个搜索选项，允许管理员输入用户 ID，然后由系统输出该用户当前正在哪个实验室的哪台计算机上登录；如果该用户 ID 当前没有登录任何一台计算机，就输出“None”。

应该为实验室使用长度为 4 的固定数组。每个数组项都指向一个动态数组，后者存储了每台计算机的用户登录信息。

下图展示了具体的结构。有时将这个结构称为不规则数组，因为列是不等长的。

Lab 数组　计算机对应的动态数组



6. 视频讲解：*Solution to Programming Project 9.6*

动态数组的一个问题在于，一旦用 new 操作符创建好数组，大小就不能改变。例如，你可能希望像使用向量那样增删数组中的记录项。本项目要求你创建一些函数，用动态数组模拟向量的行为。

首先写程序创建由 5 个字符串构成的动态数组。将你选择的 5 个姓名存储到动态数组中。接着，完成以下两个函数：

```
string* addEntry(string *dynamicArray, int &size, string newEntry);
```

这个函数新建一个动态数组，其长度比 dynamicArray 多一个元素，将 dynamicArray 的所有元素复制到新数组，在新数组的末尾添加新项，递增 size，删除 dynamicArray，返回新的动态数组。

```
string* deleteEntry(string *dynamicArray, int &size, string entryToDelete);
```

这个函数在 dynamicArray 中搜索 entryToDelete(要删除的项)。没找到就忽略请求，返回未经修改的 dynamicArray。找到就新建动态数组，其长度比 dynamicArray 小一个元素，将除 entryToDelete 之外的其他所有元素复制到新数组，删除 dynamicArray，递减 size，返回新的动态数组。

为了测试这些函数，请在数组中增删几个姓名，并输出数组的内容。必须在 main 函数中将 addEntry 或 deleteEntry 返回的数组赋还给动态数组变量。

7. 如果 C++没有内建对二维数组的支持怎么办？在这种情况下，可以自己写一些函数，用一维数组来模拟，基本思路如下。对于以下二维数组：

```
int matrix[2][3];
```

可以用以下表格来表示它。

matrix[0][0]	matrix[0][1]	matrix[0][2]
matrix[1][0]	matrix[1][1]	matrix[1][2]

该二维数组可映射到下面这个一维数组的内存存储中；上表的每一行都用连续的内存位置来存储。事实上，你的 C++编译器就是以非常相似的方式将二维数组映射到内存的。

```
int matrix1D[6];
```

matrix1D[0]	matrix1D [1]	matrix1D [2]	matrix1D [3]	matrix1D [4]	matrix1D [5]

映射关系如下：

```
matrix[0][0]存储到matrix1D[0]中
matrix[0][1]存储到matrix1D[1]中
matrix[0][2]存储到matrix1D[2]中
matrix[1][0]存储到matrix1D[3]中
matrix[1][1]存储到matrix1D[4]中
matrix[1][2]存储到matrix1D[5]中
```

请基于这个思路来完成下面这些函数的定义：

```
int* create2DArray(int rows, int columns);
```

该函数创建一维数组来模拟二维数组，返回指向一维动态数组的指针。rows 是二维数组行数，columns 是二维数组列数。返回值是指向一维动态数组的指针，该一维数组足够容下大小为 rows * columns 的二维数组。

注意 `int *ptr = create2DArray(2,3);`将创建类似于用 `int ptr[2][3];`创建的一个数组。

```
void set(int *arr, int rows, int columns, int desiredRow, int desiredColumn, int val);
```

该函数将 val 存储到模拟的二维数组的 desiredRow, desiredColumn 处。如目标行、列位置无效，函数应打印一条错误消息。

- arr 是用来模拟二维数组的一维数组。
- rows 是二维数组的总行数。
- columns 是二维数组的总列数。
- desiredRow 是基于零的行索引，是调用者想在二维数组中访问的行位置。
- desiredColumn 是基于零的列索引，是调用者想在二维数组中访问的列位置。
- val 是想要存储到 desiredRow 和 desiredColumn 处的值。

```
int get(int *arr, int rows, int columns, int desiredRow, int desiredColumn);
```

该函数返回模拟的二维数组的 desiredRow, desiredColumn 处的值。如目标行、列位置无效，函数应打印一条错误消息。

- arr 是用来模拟二维数组的一维数组。
- rows 是二维数组的总行数。
- columns 是二维数组的总列数。
- desiredRow 是基于 0 的行索引，是调用者想在二维数组中访问的行位置。
- desiredColumn 是基于 0 的列索引，是调用者想在二维数组中访问的列位置。

创建合适的测试程序来调用全部三个函数。

8. 写程序统计一项作业的成绩。程序以整数形式输入每个学生的成绩，将成绩保存到向量中(已在第 8 章讨论)。成绩可一直输入，直到输入-1 表示结束。然后，程序扫描向量来统计每个成绩的出现次数。统计时，最小的成绩是 0，但程序应该判断用户输入的最大值。用动态数组存储统计结果。将统计结果输出到控制台。例如，对于以下输入：

```
20
30
4
20
30
30
-1
```

输出应该如下所示：

```
Number of 4's: 1
Number of 20's: 2
Number of 30's: 3
```

第 10 章　定义类

"到时候了，"海象说，
"咱们来东拉西扯。
谈谈密封蜡、靴子和船；
还有皇帝和白菜。"

——*刘易斯·卡洛尔，《爱丽丝镜中奇遇记》*

概述

第 6 章介绍了如何使用类和对象，但没有谈及如何定义类。本章介绍如何定义自己的类。类是一个数据类型。可采用与使用预定义数据类型(比如 int，char 和 ifstream)相同的方式来使用自己定义的类。但除非以正确方式定义类，否则它们不具有和预定义数据类型一样良好的行为。所以，本章用大量篇幅解释了一个良好的类定义需满足哪些条件，并描述如何采用与现代编程实践一致的方式定义自己的类。

介绍类之前，首先要讲解**结构**。按我们描述的方式使用，结构相当于一种简化的类，它是帮你理解类的"跳板"。

预备知识

本章基于第 2 章～第 6 章的知识。

10.1　结　　构

第 6 章讲过，对象是有成员函数的变量，而类是数据类型，变量即对象。所以类的定义即数据类型的定义，它描述两件事情：①变量能容纳什么种类的值；②成员函数是什么。我们分两阶段完成类的定义。首先解释如何为结构提供类型定义。可将结构(下面要讨论的那种)视为无任何成员函数的对象。掌握结构后，自然就会定义类了。

用于异种数据的结构

有时需要获得由不同(异种)类型的值构成的集合，再将集合视为整体进行处理。以一份银行定期存单(Certificate of Deposit，CD)为例。CD 是银行账户，必须在指定月数之后才能提现。一份定期存单自然关联了三种数据：账户余额、利率及存期。最后一项以月为单位。前两项可表示成 double 值，月数可表示成 int 值。图 10.1 定义了名为 CDAccount 的结构，它正是为这种银行账户设计的。定义嵌套在一个完整程序中，程序演示了这个结构类型定义。如示范对话所示，银行只允许短期定存单，所以存期最长为 12 个月。下面看看这个结构具体是如何定义和使用的。

结构的定义如下所示：

```
struct CDAccount
{
    double balance;
    double interestRate;
    int term;  // 存款期限(月数)
};
```

关键字 struct 声明这是结构类型定义。标识符 CDAccount 是结构类型名称。结构类型的名称称为**结构标记**。结构标记可为任意合法标识符(但不能是关键字)。虽然 C++语言没有专门要求，但结构标记通常采用大小写字母混合的方式，而且以大写字母开头。花括号内声明的标识符是**成员名称**。在本例中，结构类型定义以花括号和分号结束。

结构定义通常放在任何函数定义的外部(类似于全局常量声明，它们也要放在所有函数定义的外部)。这样在定义好结构之后，所有代码都能使用该结构类型。

图 10.1 结构定义

```
1  // 该程序用于演示 CDAccount 结构类型
2  #include <iostream>
3  using namespace std;
4  // 一个银行定期存单结构:
5  struct CDAccount
6  {
7      double balance;
8      double interestRate;
9      int term; // 存期(月数)
10 };
11
12
13 void getData(CDAccount& theAccount);
14 // 后条件: theAccount.balance 和 theAccount.interestRate 获得用户从键盘输入的值
15
16
17 int main()
18 {
19     CDAccount account;
20     getData(account);
21
22     double rateFraction, interest;
23     rateFraction = account.interestRate/100.0;
24     interest = account.balance * rateFraction*(account.term/12.0);
25     account.balance = account.balance + interest;
26
27     cout.setf(ios::fixed);
28     cout.setf(ios::showpoint);
29     cout.precision(2);
30     cout << "When your CD matures in "
31          << account.term << " months,\n"
32          << "it will have a balance of $"
33          << account.balance << endl;
34     return 0;
35 }
36
37 // 使用 iostream:
38 void getData(CDAccount& theAccount)
39 {
40     cout << "Enter account balance: $";
41     cin >> theAccount.balance;
42     cout << "Enter account interest rate: ";
43     cin >> theAccount.interestRate;
44     cout << "Enter the number of months until maturity\n"
45          << "(must be 12 or fewer months): ";
46     cin >> theAccount.term;
47 }
```

示范对话

```
Enter account balance: $100.00
Enter account interest rate: 10.0
Enter the number of months until maturity
(must be 12 or fewer months): 6
When your CD matures in 6 months,
it will have a balance of $105.00
```

定义好结构类型后，结构类型可像预定义类型(int，char 等)那样使用。例如，下面定义两个变量，名为 myAccount 和 yourAccount，两者都是 CDAccount 类型：

```
CDAccount myAccount, yourAccount;
```

结构变量和其他任何变量一样能容纳值。**结构值**是由较小的值（称为**成员值**）构成的集合。结构定义中声明的每个成员名称都有成员值。例如，CDAccount 类型的一个值是三个成员值的集合：两个 double 类型和一个 int 类型。作为结构值的组成部分，这些成员值存储在**成员变量**中。下面将具体讨论。

每个结构类型都定义了一个成员名称列表。在图 10.1 中，CDAccount 结构有三个成员名，分别是 balance，interestRate 和 term。每个成员名都可用于访问一个较小的变量，该变量属于较大的结构变量的一部分。这些较小的变量称为成员变量。为了指定成员变量，需要先写结构变量的名称，后跟一个圆点，再写成员名。例如，假定 account 是图 10.1 声明的 CDAccount 结构类型的变量，那么结构变量 account 有以下三个成员变量：

```
account.balance
account.interestRate
account.term
```

前两个成员变量是 double 类型，最后一个是 int 类型。这些成员变量的用法和那些类型的其他任何变量无异。例如，可通过下面三个赋值语句为上述成员变量赋值：

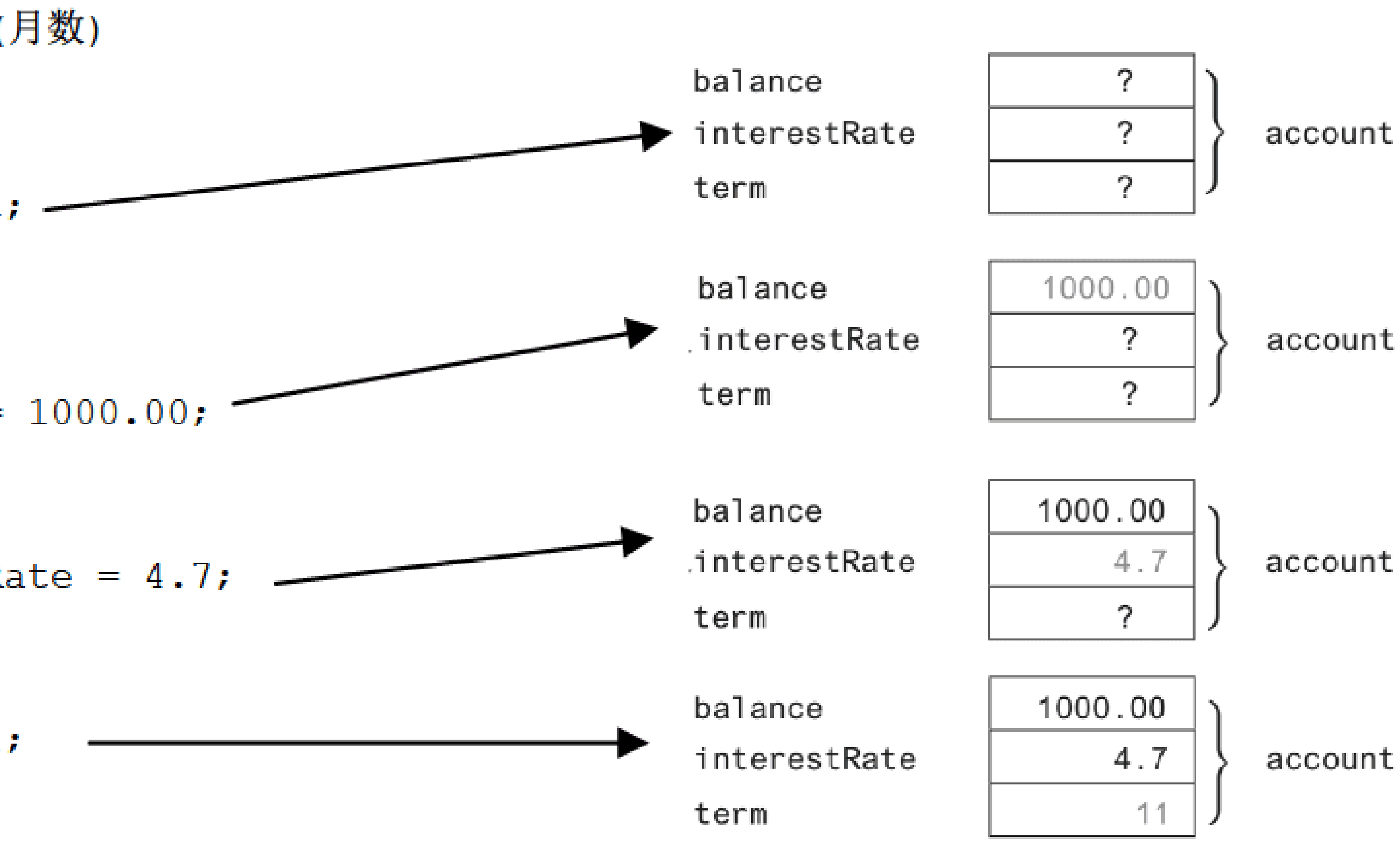
```
account.balance = 1000.00;
account.interestRate = 4.7;
account.term = 11;
```

这三个语句的结果如图 10.2 所示。成员变量可采用与普通变量一样的方式使用。例如，图 10.1 的下面这一行对成员变量 account.balance 包含的值与普通变量 interest 包含的值进行求和，结果放回成员变量 account.balance 中：

```
account.balance = account.balance + interest;
```

图 10.2　成员值

```
1   struct CDAccount
2   {
3       double balance;
4       double interestRate;
5       int term; // 存期(月数)
6   };
7   int main()
8   {
9       CDAccount account;
10          …
11
12
13      account.balance = 1000.00;
14
15
16      account.interestRate = 4.7;
17
18
19      account.term = 11;
20
21
22
```



注意，可以像第 6 章那样使用圆点操作符指定结构的成员。第 6 章是用圆点操作符指定类的成员函数。唯一的区别在于，在结构的情况下，成员是变量而非函数。

两个或更多结构类型可使用相同成员名称。例如，在同一个程序中拥有以下两个类型定义完全合法：

①
```
struct FertilizerStock // 化肥
{
    double quantity; // 数量
    double nitrogenContent;
};
```

②
```
struct CropYield // 作物
{
   int quantity; // 产量
   double size;
};
```

重复名称不会冲突。例如，假定声明以下两个结构变量：

```
FertilizerStock superGrow;
CropYield apples;
```

那么 superGrow 化肥的数量保存在成员变量 superGrow.quantity 中，而苹果产量保存在成员变量 apples.quantity 中。圆点操作符和结构变量为 quantity 赋予了不同的含义。

结构值可被视为成员值的一个集合。从这个角度看，一个结构值就是多个不同的值。结构值也可被视为一个复合值(它恰好由多个成员值构成)。由于结构值可被看成是单个值，所以在使用结构值和结构变量时，可采取和使用预定义类型(比如 int)的简单值和简单变量一样的方式，尤其是可用等号来指派结构值。例如，假定 apples 和 oranges 是刚才定义的 CropYield 类型的结构变量，那么以下赋值语句完全合法：

```
apples = oranges;
```

该赋值语句等价于以下语句：

```
apples.quantity = oranges.quantity;
apples.size = oranges.size;
```

以这种方式向结构变量赋值执行的是**浅拷贝**，即单独的成员变量被直接拷贝。简单变量没问题，但以后就会说到，动态分配的变量会出问题。

陷阱：结构定义中忘记添加分号

在结构定义中添加最后一个花括号后，你或许觉得结构定义已经完成了，但实际并非如此。还必须在最后那个花括号后添加分号。这个分号是有来历的，它涉及本书没机会用到的一个功能。结构定义不仅仅是定义，还可用于声明结构变量。可在最后一个花括号和最后那个分号之间列出结构变量。例如，下面定义一个名为 WeatherData 的结构，并声明该类型的两个结构变量 dataPoint1 和 dataPoint2：

```
struct WeatherData
{
    double temperature;
    double windVelocity;
} dataPoint1, dataPoint2;
```

但本书不准备利用这个功能。我们坚持将结构定义与该结构类型的变量声明区分开。因此，我们的结构总是在最后一个花括号后立即添加分号。■

圆点操作符

圆点操作符指定结构变量的成员变量。

语法

圆点操作符

```
Structure Variable Name.Member Variable Name
```

示例

```
struct StudentRecord
{
    int studentNumber;
    char grade;
};

int main()
{
    StudentRecord yourRecord;
    yourRecord.studentNumber = 2001;
    yourRecord.grade = 'A';
```

有的教材将圆点操作符称为成员访问操作符，我们不准备用那个术语。

结构作为函数参数

函数可以有结构类型的传值/传引用参数。例如，图 10.1 的程序定义的 getData 函数有一个 CDAccount 结构类型的传引用参数。

函数返回值也可以是结构。例如，以下函数获取 3 个参数并返回 CDAccount 值：

```
CDAccount shrinkWrap( double theBalance,
                      double theRate,
                      int theTerm )
{
   CDAccount temp;
   temp.balance = theBalance;
   temp.interestRate = theRate;
   temp.term = theTerm;
   return temp;
}
```

注意，CDAccount 类型的局部变量 temp，它构造一个完整结构值并由函数返回。一旦定义好 shrinkWrap 函数，就可以像下面这样为 CDAccount 类型的变量赋值：

```
CDAccount  newAccount;
newAccount = shrinkWrap(1000.00, 5.1, 11);
```

编程提示：使用层次化结构

有时希望结构的成员本身就是一种较小的结构。下面定义一个 PersonInfo 结构来存储一个人的身高、体重和生日：

```
struct Date
{
```

```
    int month;
    int day;
    int year;
};

struct PersonInfo
{
    double height;   // 以英寸为单位的身高
    int weight;      // 以磅为单位的体重
    Date birthday;   // 用 Date 结构记录的生日
};
```

可采取标准方式声明 PersonInfo 类型的结构变量：

```
PersonInfo person1;
```

假设结构变量 person1 已赋好值，已记录一个人的生日，就可以像下面这样将出生年份输出到屏幕：

```
cout << person1.birthday.year;
```

这种表达式要从左向右仔细看。最左边的 person1 是 PersonInfo 类型的结构变量。为了访问名为 birthday 的成员变量，要像下面这样使用圆点操作符：

```
person1.birthday
```

该成员变量本身是 Date 类型的结构变量。所以，它有自己的成员变量。为了访问结构变量 person1.birthday 的成员变量，需再添加一个圆点操作符，以及一个成员变量名称(比如 year)，这就得到了上面显示的 person1.birthday.year。 ■

简单结构类型

可以像下面这样定义**结构类型**。*StructureTag* 是结构类型名称。

语法

```
struct StructureTag
{
  Type 1 MemberVariableName1;
  Type 2 MemberVariableName2;
   . . .
  TypeLast MemberVariableNameLast;
}; <———————— 别忘了这个分号
```

示例

```
struct Automobile
{
  int year;
  int doors;
  double horsePower;
  char model;
};
```

相同类型的成员可合并成一个以逗号分隔的列表(虽然本书不准备这样做)。例如，以下定义等价于前面的结构定义：

```
struct Automobile          // 汽车
{
```

```
    int year, doors;        // 出厂年份和门数
    double horsePower;      // 马力
    char model;             // 型号
};
```

结构类型的变量可采取和其他类型的变量一样的方式来声明。例如：

```
Automobile myCar, yourCar;
```

成员变量用**圆点操作符**指定，例如 myCar.year，myCar.doors，myCar.model 以及 myCar.horsePower。

对结构进行初始化

可在声明结构的同时初始化。向结构变量赋值请在变量名后添加等号，再添加用花括号封闭的成员值列表。以上一节定义的日期结构类型为例：

```
struct Date
{
   int month;
   int day;
   int year;
};
```

定义好 Date 类型后，可以像下面这样声明并初始化名为 dueDate 的结构变量：

```
Date dueDate = {12, 31, 2004};
```

注意，初始化顺序必须与成员变量在结构类型定义中的顺序一致。本例 dueDate.month 接收第一个初始值 12；dueDate.day 接收第二个初始值 31，dueDate.year 接收第三个初始值 2004。

初始值比 struct 成员多会出错。相反，初始值比 struct 成员少就依次用提供的值来初始化数据成员。未获得初始值的数据成员初始化成与变量类型匹配的零值。

√ 自测题

1. 给定以下结构和结构变量声明：

```
struct TermAccount
{
   double balance;
   double interestRate;
   int term;
   char initial1;
   char initial2;
};
TermAccount account;
```

说明以下各项的类型，标出任何有错的。

a. account.balance
b. account.interestRate
c. TermAccount.term
d. savingsAccount.initial1
e. account.initial2

f. account

2. 给定以下类型定义：

```
struct ShoeType
{
    char style;
    double price;
};
```

基于以上结构类型定义，以下代码产生什么输出？

```
ShoeType shoe1, shoe2;
shoe1.style ='A';
shoe1.price = 9.99;
cout << shoe1.style << " $" << shoe1.price << endl;
shoe2 = shoe1;
shoe2.price = shoe2.price/9;
cout << shoe2.style << " $" << shoe2.price << endl;
```

3. 以下结构定义有什么错误？你的编译器会为这个错误显示什么消息？请用自己的话解释错在哪里。

```
struct Stuff
{
   int b;
   int c;
}
int main()
{
   Stuff x;
   // 其他代码
}
```

4. 给定以下结构定义：

```
struct A
{
   int memberB;
   int memberC;
};
```

请声明 x，让它具有这种结构类型。同时，将 x 的成员 memberB 和 memberC 分别初始化成值 1 和 2。**注意**：要求的是初始化，而不是成员赋值。两者的区别非常重要，本书后面会详述。

5. 下面是一个结构类型的初始化代码。指出每次初始化的结果。说明这些初始化存在的任何问题。

```
struct Date
{
    int month;
    int day;
    int year;
};
```

a. `Date dueDate = {12, 21};`
b. `Date dueDate = {12, 21, 2022};`
c. `Date dueDate = {12, 21, 20, 22};`
d. `Date dueDate = {12, 21, 22};`

6. 为员工记录定义结构类型。这些记录包括一个员工的"工资"(wage rate)；"应休假"(accrued vacation)，这是一个整数天数；以及"身份"(status)，可选择计时(hourly)或领薪(salaried)。将身份表示成两个 char 值之一：'H'或'S'。将类型命名为 EmployeeRecord。

7. 给出与以下函数声明对应的函数定义(ShoeType 类型已在自测题 2 中给出)。

```
void readShoeRecord(ShoeType& newShoe);
// 使用从键盘读入的值来填充 newShoe
```

8. 给出与以下函数声明对应的函数定义(ShoeType 类型已在自测题 2 中给出)。

```
ShoeType discount(ShoeType oldRecord);
// 返回与它的实参相同的一个结构，但价格降低 10%
```

9. 定义 StockRecord 结构类型。有两个成员变量，一个是 shoeInfo，具有自测题 2 中给出的 ShoeType 类型；另一个是 arrivalDate，具有自测题 5 给出的 Date 类型。

10. 声明 StockRecord 类型(在上题给出)的一个变量，并写一个语句将 arrivalDate(到货日期)中的年份(year)设为 2006。

10.2　类

我不关心任何俱乐部接纳我为会员这件事情。[①]

——格罗克·马克斯[②](1890—1977)，《格罗克信件》

定义类和成员函数

类是数据类型，其变量就是对象。第 6 章讲过，**对象**是变量，其中除了有成员函数，还有数据值。[③]所以在 C++程序中，类的定义应该是数据类型定义，它描述变量能容纳哪些种类的值，而且有哪些成员函数。结构定义也能描述部分内容。结构是定义好的类型，允许定义成员变量来定义结构类型的值。要从结构获得类，添加成员函数即可。

图 10.3 是一个示例类定义。DayOfYear 类的对象容纳了日期值，比如 January 1 或 July 4。可用这些值记录节假日、生日和其他纪念日。在 DayOfYear 的定义中，月份作为 int 值来记录，1 表示 1 月，2 代表 2 月，依此类推。一月中的天数在另一个 int 成员变量中记录。DayOfYear 类包含无参成员函数 output，能将月数和天数值输出到屏幕。下面详细解释 DayOfYear 类的定义。

图 10.3　包含成员函数的类

```
1  // 该程序用于演示一个非常简单的类
2  // DayOfYear 类的一个更好的版本在图 10.4 给出
3  #include <iostream>
4  using namespace std;

5  class DayOfYear
6  {
7  public:
8      void output();   ◄——— 成员函数声明
9      int month;
10     int day;
11 };
12 int main()
13 {
14     DayOfYear today, birthday;

15     cout << "Enter today's date:\n";
16     cout << "Enter month as a number: ";
```

① 会员和成员的英语都是 member。——译注

② 朱利叶斯·亨利·格劳乔·马克斯(Julius Henry Groucho Marx)，美国喜剧演员与电影明星。他以机智问答及比喻闻名，与家族成员(马克思兄弟)合作拍摄了 15 部电影，并且“单飞”成绩也十分耀眼，以担任广播及电视节目 *You Bet Your Life* 主持人而闻名。马克斯的特色十分显著：显眼的胡须、眉毛以及眼镜。——编注

③ 对象实际是变量的值，而非变量本身。但由于我们用变量来命名它容纳的值，所以可以简化讨论，不必过于计较这个差异，把变量和它的值看成是一回事。

```
17      cin >> today.month;
18      cout << "Enter the day of the month: ";
19      cin >> today.day;
20      cout << "Enter your birthday:\n";
21      cout << "Enter month as a number: ";
22      cin >> birthday.month;
23      cout << "Enter the day of the month: ";
24      cin >> birthday.day;
25      cout << "Today's date is ";
26      today.output();                     ← 对成员函数 output 的调用
27      cout << "Your birthday is ";
28      birthday.output();
29      if (today.month == birthday.month
30         && today.day == birthday.day)
31         cout << "Happy Birthday!\n";
32      else
33         cout << "Happy Unbirthday!\n";
34      return 0;
35  }
36  // 使用 iostream:
37  void DayOfYear::output()                ← 成员函数定义
38  {
39      cout << "month = " << month
40          << ", day = " << day << endl;
41  }
```

示范对话

```
Enter today's date:
Enter month as a number: 10
Enter the day of the month: 15
Enter your birthday:
Enter month as a number: 2
Enter the day of the month: 21
Today's date is month = 10, day = 15
Your birthday is month = 2, day = 21
Happy Unbirthday!
```

DayOfYear 类的定义在图 10.3 中靠近顶部的位置。暂时忽略包含关键字 public 的那一行。该行指出成员变量和函数没有限制。本章后面将具体解释这一行的作用。DayOfYear 类其余的定义和结构定义很相似，只是使用了关键字 class，而不是 struct，而且列出了成员函数 output(以及成员变量 month 和 day)。注意，在列出成员函数 output 时，只给出了它的函数声明。至于成员函数的具体定义，则在其他地方给出(在 C++类定义中，成员变量和成员函数可按任意顺序混合。但我们遵循的风格是“成员函数先于成员变量列出”)。声明类类型的对象(也就是变量)时，采用的方式和声明预定义类型(比如 int)的变量一样，也和结构变量的声明一样。

第 6 章讲过如何调用预定义类的成员函数。调用为类定义的成员函数时，采用的方式是一样的。例如，图 10.3 的程序采用以下方式声明 DayOfYear 类型的两个对象：

```
DayOfYear today, birthday;
```

要为 today 对象调用成员函数 output，采用以下方式：

```
today.output();
```

要为 birthday 对象调用成员函数 output，采用以下方式：

```
birthday.output();
```

定义成员函数时，定义中必须包括类名，因为可能两个或者更多的类具有同名的成员函数。图 10.3 只有一个类定义，但在其他时候可能有多个类定义，而且每个类都可能有名为 output 的成员函数。图 10.3 显示了 DayOfYear 类的成员函数 output 的定义。该定义类似于普通的函数定义，但仍有一些区别。

下面是成员函数 output 的函数定义头：

```
void DayOfYear::output()
```

操作符::是**作用域解析操作符**，作用和圆点操作符相似。无论圆点操作符，还是作用域解析操作符，都指出一个成员函数具体是谁的成员。但作用域解析操作符::作用于类名，而圆点操作符作用于对象(类的变量)。作用域解析操作符必须是连续两个冒号，中间不能有空格。操作符前的类名为**类型限定符**，它使函数名具体化，指出该函数从属（限定）于哪个类型。

成员函数定义

成员函数采用和其他函数一样的方式定义，只是要在函数头中给出 *ClassName* 和作用域解析操作符::。

语法

```
ReturnedType ClassName::FunctionName(ParameterList)
{
    函数主体语句
}
```

示例

```
// 使用 iostream:
void DayOfYear::output()
{
    cout << "month = " << month
         << ", day = " << day << endl;
}
```

DayOfYear 类的定义已在图 10.3 中给出，month 和 day 是 DayOfYear 类的成员变量。注意，month 和 day 没有附加对象名和圆点前缀。

接着看成员函数 DayOfYear::output 的定义。定义中直接使用成员名称 month 和 day，没有先给出对象名和圆点操作符。这并不奇怪。现在只是定义成员函数 output。这个 output 定义应用于 DayOfYear 类型的所有对象，但目前不知道准备使用的 DayOfYear 类型的对象的名称，所以无法在此给出名称。一旦调用成员函数，就像下面这样：

```
today.output();
```

函数定义中的所有成员名都会用调用对象(这里是 today)的名称来具体化。所以，上述函数调用等价于：

```
{
  cout << "month = " << today.month
       << ", day = " << today.day << endl;
}
```

在成员函数定义中，是直接使用类的所有成员(无论数据成员还是函数成员)的名称，不要使用圆点操作符。

圆点操作符和作用域解析操作符

两个操作符都随成员名称一起使用，指出它们具体是谁的成员。例如，假设声明一个名为 DayOfYear 的类，并像下面这样声明它的一个名为 today 的对象：

```
DayOfYear today;
```

那么要用**圆点操作符**指定 today 对象的成员。例如，output 是 DayOfYear 类的成员函数(在图 10.3 中定义)，以下函数调用输出存储在 today 对象中的数据值：

```
today.output();
```

为成员函数给出函数定义时，则要使用**作用域解析操作符::**指定类名。例如，成员函数 output 的函数定义头是：

```
void DayOfYear::output()
```

记住，作用域解析操作符::用于类名，而圆点操作符用于那个类的对象。

√ 自测题

11. 下面重新定义了图 10.3 的 DayOfYear，新增了名为 input 的成员函数，请为该函数写合适的定义。

```
class DayOfYear
{
public:
    void input();
    void output();
    int month;
    int day;
};
```

12. 给定以下类定义，为成员函数 set 写合适的定义：

```
class Temperature
{
public:
    void set(double newDegrees, char newScale);
    // 将成员变量设为作为实参传递的值

    double degrees;
    char scale;  // 'F'表示 Fahrenheit(华氏度), 'C'代表 Celsius(摄氏度)
};
```

13. 请清楚地描述“圆点操作符.”和“作用域解析操作符::”在含义及用法上的区别。

公共成员和私有成员

预定义类型(比如 double)不是作为 C++类来实现的，但 C++编译器的作者确实设计了一些方式在计算机中表示 double 类型的值。double 类型可用多种方式实现。事实上，不同版本的 C++的确采用了稍有不同的方式实现 double 类型。但将 C++程序从一台计算机转移到以不同方式实现 double 类型的另一台计算机，程序应该还是能正确工作。[①]类是你

① 理想情况下如此。对于简单程序，这一点可以保证。

定义的类型，它们的行为应该和预定义类型一样。可生成一个库，在其中容纳自己的类类型定义，并将这些类型当作预定义类型来使用。例如，可将每个类定义都放到一个单独的文件中，再将它复制到任何使用这些类型的程序中。

和预定义类型一样，类定义应该严格区分类的使用规则以及类的实现细节。可以自由更改类的实现细节 (例如，更改成员函数定义，使其运行得更快)，但不必更改程序的其他任何部分。为了做到这一点，要了解类定义的另一个功能。

重新分析图 10.3 的 DayOfYear 类型定义。DayOfYear 类型设计用于容纳日期值，比如生日和节假日。用两个整数表示这些日期，一个表示月份，另一个表示天数。但未来某一天，可能决定将月份的表示从 int 类型的 1 个变量更改为 char 类型的 3 个变量。在修改版本中，3 个字符可能是月份名称的缩写。例如，3 个 char 值'J'，'a'和'n'表示 January。但无论使用 int 类型的 1 个成员变量来记录月份，还是使用 char 类型的 3 个成员变量，都属于实现细节的范畴。使用 DayOfYear 类型的程序员不应关心这个问题。当然，一旦更改了 DayOfYear 类的月份表示方式，就必须更改成员函数 output 的实现——但那应该是你唯一需要更改的东西。在程序中不应更改使用 DayOfYear 类定义的其他任何部分。遗憾的是，图 10.3 的程序不符合这一理想。例如，一旦将名为 month 的 1 个成员变量替换成 char 类型的 3 个成员变量，名为 month 的成员变量就不复存在，所以必须更改程序中执行输入的那些部分，if-else 语句也要改。

理想的类定义应该能自由更改类的实现细节。在使用该类的任何程序中，唯一要更改的就是成员函数的定义。为此，必须拥有数量足够多的函数，确保永远都不需要直接访问成员变量，而是始终通过成员函数来访问。这样一来，一旦需要更改成员变量，就只需更改成员函数的定义，使之匹配成员变量的变动，程序中的其他任何东西都不必更改。图 10.4 重新定义了 DayOfYear 类，使其包含足够多的成员函数，让它们做我们希望程序做的所有事情。总之，要保证程序不直接引用任何成员变量。仔细分析图 10.4，发现只有成员函数的定义才使用了成员变量名称 month 和 day。在成员函数的定义之外，没有任何一个地方引用了 today.month，today.day，bachBirthday.month 以及 bachBirthday.day。

图 10.4　包含私有成员的类

```
1  // 该程序用于演示 DayOfYear 类
2  #include <iostream>
3  using namespace std;

4  class DayOfYear
5  {
6  public:
7      void input();
8      void output();

9      void set(int newMonth, int newDay);
10     // 前条件：newMonth 和 newDay 构成一个可能的日期
11     // 后条件：日期根据传递的实参来重置

12     int getMonth();
13     // 返回月份，1 代表 1 月，2 代表 2 月，依此类推
14     int getDay();
15     // 返回天数
16 private:
17     void checkDate();
18     int month;
19     int day;
20 };
```

这是图 10.3 的 DayOfYear 类的增强版本

私有成员函数

私有成员变量

```
21  int main()
22  {
23     DayOfYear today, bachBirthday;
24     cout << "Enter today's date:\n";
25     today.input();
26     cout << "Today's date is ";
27     today.output();

28     bachBirthday.set(3, 21);
29     cout << "J. S. Bach's birthday is ";
30     bachBirthday.output();

31     if ( today.getMonth() == bachBirthday.getMonth() &&
32          today.getDay() == bachBirthday.getDay() )
33          cout << "Happy Birthday Johann Sebastian!\n";
34     else
35          cout << "Happy Unbirthday Johann Sebastian!\n";
36     return 0;
37  }
38  // 使用 iostream:
39  void DayOfYear::input()
40  {
41     cout << "Enter the month as a number: ";
42     cin >> month;
43     cout << "Enter the day of the month: ";
44     cin >> day;
45     checkDate();
46  }
47
48  void DayOfYear::output()
    <DayOfYear::output 剩余的定义与图 10.3 相同>
49
50  void DayOfYear::set(int newMonth, int newDay)
51  {
52     month = newMonth;
53     day = newDay;
54     checkDate();
55  }
56
57  void DayOfYear::checkDate()
58  {
59     if ((month < 1) || (month > 12) || (day < 1) || (day > 31))
60     {
61        cout << "Illegal date. Aborting program.\n";
62        exit(1);
63     }
64  }
65
66  int DayOfYear::getMonth()
67  {
68     return month;
69  }
70
71  int DayOfYear::getDay()
72  {
73     return day;
74  }
```

私有成员可在成员函数的定义中使用（不可在其他任何地方使用）

成员函数 input 的更理想的定义中，会在用户输入错误日期时要求用户重新输入正确日期

成员函数 checkDate 不会检查所有非法日期，但通过加长它的定义，很容易进行完整检查。详情参见自测题 14

函数 exit 已在第 6 章讨论，用于退出程序

示范对话

```
Enter today's date:
Enter the month as a number: 3
Enter the day of the month: 21
Today's date is month = 3, day = 21
J. S. Bach's birthday is month = 3, day = 21
Happy Birthday Johann Sebastian!
```

图 10.4 的程序实现了一个新功能，它确保使用 DayOfYear 类的任何程序员都无法直接引用它的任何成员变量。在 DayOfYear 类的定义中，注意包含关键字 private 的那一行。这一行后列出的所有成员都是**私有成员**；换言之，只能在成员函数定义中访问，在其他任

何地方都无法访问。在程序 main 部分，或者在非该类成员函数的其他函数中，试图访问其中一个成员变量，编译器将报错。在成员变量和成员函数列表中插入关键字 private 和一个冒号，private:标签之后的所有成员都会成为私有成员。private:标签之后的变量称为私有成员变量，函数称为私有成员函数。

对于图 10.4 的 DayOfYear 类，它的所有成员变量都是私有成员。私有成员变量可在任何成员函数的定义中使用，但在其他任何地方都不能使用。例如，根据修改过的 DayOfYear 类定义，以下两个赋值语句在程序的 main 部分是非法的：

```
DayOfYear today;    // 这一行合法
today.month = 12;   // 非法
today.day = 25;     // 非法
```

除非在成员函数的定义中，否则对这些私有变量的任何引用都属于非法。由于 month 和 day 是私有成员变量，所以在任何程序的 main 部分，假定将 today 声明为 DayOfYear 类型的对象，那么以下语句也是非法的：

```
cout << today.month;   // 非法
cout << today.day;     // 非法
if (today.month == 1)  // 非法
   cout << "January";
```

一旦将成员变量设为私有成员变量，除非使用某个成员函数，否则没有办法更改它的值(或以其他任何方式引用该变量)。这是严格且合理的限制。将所有成员变量变成私有，程序员更容易理解和更新自己的代码。

图 10.4 的程序表面上没有真正禁止直接访问私有成员变量，因为可以使用成员函数 DayOfYear::set 更改它们，还可以使用成员函数 DayOfYear::getMonth 以及 DayOfYear::getDay 获取它们的值。这对图 10.4 的程序来说似乎是成立的，但一旦更改了日期中的月份和/或天数的表示方式，这方面的疑虑就可以减轻了。例如，假定像下面这样更改 DayOfYear 的类型定义：

```
class DayOfYear
{
public:
    void input();
    void output();

    void set(int newMonth, int newDay);
    // 前条件：newMonth 和 newDay 构成一个可能的日期
    // 后条件：日期根据传递的实参来重置

    int getMonth();
    // 返回月份，1 代表 1 月，2 代表 2 月，依此类推

    int getDay();
    // 返回天数
private:
    void DayOfYear::checkDate();
    char firstLetter; // 月份名称第一个字母
    char secondLetter; // 月份名称第二个字母
    char thirdLetter; // 月份名称第三个字母
    int day;
};
```

这会使成员函数复杂一些，但通过重新定义成员函数，可以保证其行为不变。例如，getMonth 函数的定义可以像下面这样开头：

```
int DayOfYear::getMonth()
{
    if (firstLetter == 'J' && secondLetter == 'a'
        && thirdLetter == 'n')
      return 1;
    if (firstLetter == 'F' && secondLetter == 'e'
        && thirdLetter == 'b')
      return 2;
...
```

虽然麻烦，但不难。

在成员函数 DayOfYear::set 和 DayOfYear::input 中，我们进行了一次检查，确保成员变量 month 和 day 被设为合法的值。这个检查是通过调用成员函数 DayOfYear::checkDate 来完成的。如成员变量 month 和 day 是公共而非私有，这些成员变量就能被设置成任意值，其中包括非法值。将成员变量设为私有，并只通过成员函数来处理，就可确保成员变量永远不会被设置成非法或无意义的值(自测题 14 要求重新定义成员函数 DayOfYear::checkDate，使其对非法日期执行一个完整的检查)。

封 装

封装也称为**数据隐藏**，目的是使类中的变量私有，只能由公共或受保护函数通过一个受控制的接口访问。这样就可自由修改内部变量和数据结构，而不影响使用类的代码。

也可将成员函数变成私有。与私有成员变量相似，私有成员函数能在其他任何成员函数的定义中使用，但在除此之外的其他任何地方(比如在使用了该类的一个程序的 main 部分中)都不能使用。例如，图 10.4 的成员函数 DayOfYear::checkDate 就是私有成员函数。根据经验，如果一个成员函数只在其他成员函数的定义中作为辅助函数使用，就将其设为私有。

视频讲解：*Class Scope, Public and Private Members*

关键字 private 指定私有成员；类似地，关键字 public 指定**公共成员**。例如，对于图 10.4 的 DayOfYear 类，除了 DayOfYear::checkDate 之外的所有成员函数都是公共成员(所有成员变量都是私有成员)。公共成员可在程序的 main 主体中使用，或在任何函数(包括非成员函数)的定义中使用。

在类定义中，public 和 private 可以多次出现。每次插入标签 public:就会将成员列表从私有更改为公共。每次插入标签 private:成员列表就会变回私有。

例如在以下结构定义中，成员函数 doSomethingElse 和成员变量 moreStuff 是私有成员，其他 4 个成员都是公共成员：

```
class SampleClass
{
public:
    void doSomething();
```

```
    int stuff;
private:
    void doSomethingElse();
    char moreStuff;
public:
    double doYetAnotherThing();
    double evenMoreStuff;
};
```

在类定义起始位置列出成员，不在这些成员之前插入 public:或 private:时，它们默认为私有成员。但最好总是将每一组成员都显式标记为 public 或 private。

类 和 对 象

类是变量为对象的类型。对象可拥有成员变量和成员函数。类定义的语法如下。

语法

```
class ClassName
{
public:
    MemberSpecification 1        ← 公共成员
    MemberSpecification 2
       ⋮
    MemberSpecification n
private:
    MemberSpecification n+1      ← 私有成员
    MemberSpecification n+2
       ⋮
};  ← 不要遗漏这个分号
```

每个 *MemberSpecification_i* 要么是成员变量声明，要么是成员函数声明(可添加更多的 public 和 private 小节)。

示例

```
class Bicycle
{
public:
    char getColor();
    int numberOfSpeeds();
    void set(int theSpeeds, char theColor);
private:
    int speeds;
    char color;
};
```

定义好类之后，可以像声明其他任何类型的变量那样声明它的对象(记住对象是“类”类型的变量)。例如，以下语句声明 Bicycle 类型的两个对象：

```
Bicycle myBike, yourBike;
```

编程提示：将所有成员变量设为私有

定义类时，通常将所有成员变量设为私有。这意味着只能使用成员函数来访问或更改成员变量。本章准备用大量篇幅来解释以这种方式定义类的原因和做法。 ■

编程提示：定义取值函数和赋值函数

操作符==能测试简单类型的两个值是否相等。遗憾的是，预定义操作符==不能自动应用于对象。第 11 章会介绍如何将操作符==应用于自定义类的对象。但在此之前，暂时无法将相等操作符==用于对象(和结构)。这会带来一定的麻烦。定义类时，首选的风格是将所有成员变量都设为私有。所以，为了测试两个对象是否代表相同的值，需要采取某种方式访问成员变量的值(或者与成员变量的值等价的东西)。这样一来，就可通过测试成员变量的值来测试对象是否相等。为了在图 10.4 中进行这个操作，我们的办法是在 if-else 语句中使用成员函数 getMonth 和 getDay。

取值函数是访问私有成员变量值的成员函数(比如 getMonth 和 getDay)。基于我们迄今为止学到的编程技术，最好是为每个类都包括一套完整的取值函数，这样才能测试对象的相等性。取值函数不需要按字面意义返回每个成员变量的值，但必须返回与那些值等价的东西。第 11 章将用一种更优雅的技术测试两个对象是否相等；但即使在学了那种技术之后，取值函数仍能带来某些方便。

赋值函数是允许更改私有成员变量值的成员函数(比如图 10.4 的 set)。最好为每个类都包括相应的赋值函数，以便更改对象中存储的数据。 ■

取值函数和赋值函数

获取类的私有成员变量值的成员函数称为**取值函数**。取值函数不需要返回每个成员变量的字面值，但必须返回与那些值等价的东西。虽然 C++语言规范没有要求，但通常都在取值函数的名称中包括单词“get”。

更改类的私有成员变量值的成员函数称为**赋值函数**。虽然 C++语言规范没有要求，但通常都在赋值函数的名称中包括单词“set”。

最好为每个类定义都包括相应的取值函数和赋值函数，以便存取对象中的数据。

√ 自测题

14. 图 10.4 的私有成员函数 DayOfYear::checkDate 会忽视一些非法日期，比如 February 30(2 月 30 日)。请重新定义成员函数 DayOfYear::checkDate，在发现任何非法日期后就终止程序。注意 February(2 月)可以有 29 天，这样才能支持闰年(**提示**：虽然有点儿麻烦，而且函数定义会变得比较长，但整个过程并不难)。

15. 假定程序包括以下类定义：

```
class Automobile // 汽车
{
public:
    void setPrice(double newPrice);
    void setProfit(double newProfit);
    double getPrice();
private:
    double price;
    double profit;
    double getProfit();
};
```

并假定程序 main 部分包含以下声明，而且已设置好所有成员变量的值：

```
Automobile Hyundai, jaguar;
```

那么，以下哪些语句允许在程序 main 部分中使用？

```
hyundai.price = 4999.99;
jaguar.setPrice(30000.97);
double aPrice, aProfit;
aPrice = jaguar.getPrice();
aProfit = jaguar.getProfit();
aProfit = hyundai.getProfit();
if (hyundai == jaguar)
   cout << "Want to swap cars?";
hyundai = jaguar;
```

16. 假定修改自测题 15，在 Automobile 类的定义中删除包含关键字 private 的那一行。那么，自测题 15 的答案应该如何修改？
17. 解释 public:和 private:在类定义中的作用。尤其要解释为什么不干脆将一切都变成 public:，从而避免访问时的麻烦。
18. a. 一个类要想真正有用，其中必须有多少个 public:小节？
    b. 一个类中必须有多少个 private:小节？
    c. 在类的起始花括号后直接列出成员，不先插入 public:或 private:标签，这些成员是什么成员？
    d. 在结构的起始花括号后直接列出成员，不先插入 public:或 private:标签，这些成员是什么成员？

编程提示：将赋值操作符用于对象

为对象或结构使用赋值操作符=完全合法。例如，假定像图 10.4 那样定义 DayOfYear 类，使其包含两个私有成员变量 month 和 day，并假定像下面这样声明 dueDate 和 tomorrow 对象：

```
DayOfYear dueDate, tomorrow;
```

那么以下语句完全合法(只要 tomorrow 对象的成员变量已被赋值)：

```
dueDate = tomorrow;
```

以上赋值语句等价于以下语句：

```
dueDate.month = tomorrow.month;
dueDate.day = tomorrow.day;
```

此外，即使 month 和 day 是 DayOfYear 类的私有成员变量，以上操作也合法。[①] 对象以这种方式拷贝称为浅拷贝，因为只有直接成员变量才会被拷贝。如成员变量是引用，则拷贝指向同一个引用。■

编程实例　BankAccount 类(版本 1)

图 10.5 包含银行账户的类定义，展示了你迄今为止学过的所有类定义要点。这种银行账户允许随时提取存款，和前面讲过的 CDAccount 类型不同，没有存期(term)限制。更重要的是，BankAccount 类为程序可能采取的所有操作提供了成员函数。BankAccount 类的对象有两个私有成员变量：一个记录账户余额，另一个记录利率。下面讨论 BankAccount 的一些特点。

① 第 11 章会讲到，某些情况下要为类重新定义(重载)赋值操作符=。

图 10.5 BankAccount 类

```
1  // 该程序用于演示 BankAccount 类
2  #include <iostream>
3  using namespace std;

4  // 定义银行账户的类:
5  class BankAccount
6  {
7  public:
8      void set(int dollars, int cents, double rate);
9      // 后条件: 将余额设为$dollars.cents, 将利率设为百分之 rate

10     void set(int dollars, double rate);
11     // 后条件: 将余额设为$dollars.00, 将利率设为百分之 rate

12     void update();
13     // 后条件: 将 1 年的单利添加到账户余额上

14     double getBalance();
15     // 返回当前账户余额

16     double getRate();
17     // 将当前账户利率作为一个百分比返回

18     void output(ostream& outs);
19     // 前条件: 如果 outs 是一个文件输出流, 那么 outs 已经连接到一个文件
20     // 后条件: 账户余额和利率写入 outs 流
21 private:
22     double balance;
23     double interestRate;
24
25     double fraction(double percent);
26     //将百分比转换成小数, 例如, fraction(50.3)返回 0.503
27 };

28 int main()
29 {
30     BankAccount account1, account2;
31     cout << "Start of Test:\n";
32     account1.set(123, 99, 3.0);
33     cout << "account1 initial statement:\n";
34     account1.output(cout);

35     account1.set(100, 5.0);
36     cout << "account1 with new setup:\n";
37     account1.output(cout);

38     account1.update();
39     cout << "account1 after update:\n";
40     account1.output(cout);

41     account2 = account1;
42     cout << "account2:\n";
43     account2.output(cout);
44     return 0;
45 }
46
47 void BankAccount::set(int dollars, int cents, double rate)
48 {
49     if ((dollars < 0) || (cents < 0) || (rate < 0))
50     {
51         cout << "Illegal values for money or interest rate.\n";
52         exit(1);
53     }

54     balance = dollars + 0.01*cents;
55     interestRate = rate;
56 }
57
58 void BankAccount::set(int dollars, double rate)
59 {
60     if ((dollars < 0) || (rate < 0))
```

重载成员函数 set

调用重载的成员函数 set

重载的成员函数 set 的定义

```
61      {
62          cout << "Illegal values for money or interest rate.\n";
63          exit(1);
64      }
65      balance = dollars;
66      interestRate = rate;
67  }
68
69
70  void BankAccount::update()
71  {
72      balance = balance + fraction(interestRate)*balance;   ←在成员函数定义中，可像这样
73  }                                                           调用另一个成员函数
74
75  double BankAccount::fraction(double percentValue)
76  {
77      return (percentValue / 100.0);
78  }
79
80  double BankAccount::getBalance()
81  {
82      return balance;
83  }
84
85  double BankAccount::getRate()
86  {
87      return interestRate;
88  }
89
90  // 使用 iostream:
91  void BankAccount::output(ostream& outs)  ←这个流参数接受 cout 或文件输出流
92  {
93      outs.setf(ios::fixed);
94      outs.setf(ios::showpoint);
95      outs.precision(2);
96      outs << "Account balance $" << balance << endl;
97      outs << "Interest rate " << interestRate << "%" << endl;
98  }
```

示范对话

```
Start of Test:
account1 initial statement:
Account balance $123.99
Interest rate 3.00%
account1 with new setup:
Account balance $100.00
Interest rate 5.00%
account1 after update:
Account balance $105.00
Interest rate 5.00%
account2:
Account balance $105.00
Interest rate 5.00%
```

注意，BankAccount 类有一个名为 fraction 的私有成员函数。由于私有，所以不能在 main 的主体中调用，也不能在不是 BankAccount 类的成员函数的其他任何函数体中调用。只能在 BankAccount 类的其他成员函数定义中调用 fraction 函数。之所以设计这个(以及其他任何)私有成员函数，唯一的目的就是帮助我们定义类的其他成员函数。在 BankAccount 类的定义中包括成员函数 fraction 后，就可以在 update 函数的定义中使用它。fraction 函数获取一个参数，也就是一个百分比数字，比如 10.0 代表 10.0%，并将其转换成小数，比如 0.10。这样就可根据任意百分比来计算账户的利息金额。如账户包含 100.00 美元，利率是 10%，则利息等于 100 美元乘以 0.10，即 10.00 美元。

在程序 main 主体中调用公共成员函数(比如 update)时，必须包括对象名和一个圆点，例如图 10.5 的下面这行代码：

```
account1.update();
```

但在另一个成员函数的定义中调用私有成员函数(或其他任何成员函数)时，只需使用成员函数名称，不必添加任何调用对象或圆点操作符。例如，成员函数 BankAccount::update 的定义中包括对 BankAccount::fraction 的一个调用(参见图 10.5)：

```
void BankAccount::update()
{
   balance = balance + fraction(interestRate)* balance;
}
```

fraction 函数和两个成员变量(balance 和 interestRate)的调用对象在 update 函数被调用时确定。例如以下语句：

```
account1.update();
```

它实际的含义如下：

```
{
  account1.balance = account1.balance +
  account1.fraction(account1.interestRate)*account1.balance;
}
```

注意，在调用成员函数 fraction 时，会采取与引用成员变量时一样的处理方式(也就是现场确定调用对象)。

和以前讨论的类一样，BankAccount 类也有一个成员函数能输出存储在对象中的数据。本程序将输出发送到屏幕。但在写类定义时，我们希望它适应更多使用场合，能在其他程序中不加更改地使用。由于一些程序可能希望将输出发送到文件，所以为成员函数 output 指定了一个 ostream 类型的形参，使 output 函数在调用时，可接收 cout 流或文件输出流作为实参。示例程序希望输出到屏幕，所以对 output 的第一个函数调用如下：

```
account1.output(cout);
```

其他 output 调用也用 cout 作为实参，所以全部输出都发送到屏幕。要输出到文件，首先必须将文件连接到一个输出流(第 6 章解释了具体做法)。假定文件输出流是 fout，而且已连接到一个文件，那么以下语句就将 account1 对象的数据写入文件而不是屏幕：

```
account1.output(fout);
```

BankAccount 类型的对象代表一个银行账户，它有一些余额，并支付一些利息。可用成员函数 set 设置余额和利率。注意我们重载了名为 set 的成员函数，所以现在有两个版本的 set。一个版本有三个形参，另一个只有两个。两个版本都包含一个代表利率的 double 形参。但两个版本的 set 使用不同形参来设置账户余额。一个版本用两个形参设置余额；其中，一个参数设置账户余额中的美元数；另一个设置美分数。另一个版本只用一个形参设置余额，它提供账户中的美元数，假定美分数是零。第二个版本的 set 方便，因大多数人在开户时都存入整数金额，如 1000 美元，不带美分零头。注意，这只是一个普通的重载操作。成员函数在重载时，采取的方式和普通函数的重载方式一样。

总结类的一些特征

类具有结构的所有特征，还具有与成员函数有关的所有特征。下面总结了使用类时应注意的要点。

- 类同时具有成员变量和成员函数。
- 成员(无论成员变量还是成员函数)可为公共或私有。
- 通常，类的所有成员变量都应标记为私有成员。
- 类的私有成员只能在同一个类的另一个成员函数的定义中使用。
- 类的成员函数的名称可像普通函数名称那样重载。
- 一个类可将另一个类作为自己的成员变量的类型使用。
- 函数形参可以是类类型(参考自测题 19 和自测题 20)。
- 函数可返回对象；换言之，类可以是函数返回值的类型(参考自测题 21)。

比较结构和类

如所有成员变量都公共，无成员函数，就通常使用结构。但 C++结构也可包含私有成员变量和公共/私有成员函数。除了表示法的一些区别，C++的结构能做类能做的任何事情。这一技术细节是我必须向你指明的，否则就违背了“禁止不实广告”的原则。但在理解了之后，我建议你忘记结构的这一技术细节。如严肃对待这种技术细节，非要采取和类一样的方式使用结构，那么同一个概念就会有两个名称(但具有不同的语法规则)。相反，如果像本书正文中描述的那样使用结构，就可以人为地对结构和类进行有意义的区分。与此同时，你的编程思路也会和其他大多数程序员一致。

√ 自测题

19. 为具有以下函数声明的函数给出定义(BankAccount 类已在图 10.5 定义)：

```
double difference(BankAccount account1, BankAccount account2);
// 前条件：account1 和 account2 已被赋值
// (也就是说，它们的成员变量已被赋值)
// 返回 account1 中的余额减去 account2 中的余额
```

20. 为具有以下函数声明的函数给出定义(BankAccount 类已在图 10.5 定义)：

```
void doubleUpdate(BankAccount& theAccount);
// 前条件：theAccount 已被赋值
// (也就是说，它的成员变量已被赋值)
// 后条件：账户余额被修改，
// 将 2 年的利息添加到账户上
```

21. 为具有以下函数声明的函数给出定义(BankAccount 类已在图 10.5 中定义)：

```
BankAccount newAccount(BankAccount oldAccount);
// 前条件：oldAccount 已被赋值(也就是说，它的成员变量已被赋值)
// 后条件：返回一个新的 BankAccount 对象，其余额为零，利率与 oldAccount 相同
```

例如，在定义好这个函数之后，程序中可能包含以下语句：

```
BankAccount account3, account4;
account3.set(999, 99, 5.5);
```

```
account4 = newAccount(account3);
account4.output(cout);
```

输出结果如下：

```
Account balance $0.00
Interest rate 5.50%
```

用于初始化的构造函数

声明对象时，经常需要初始化它的部分或全部成员变量。本书以后会讲到，还可能采取其他初始化操作，但“初始化成员变量”是最常见的一种初始化操作。C++为这种初始化准备了特殊机制。定义类时，可定义一种特殊的成员函数，称为**构造函数**。构造函数在声明类的对象时自动调用。构造函数通常初始化成员变量的值，并进行其他初始化操作。可像定义其他任何成员函数那样来定义构造函数，只是注意以下两点。

其一，构造函数必须与类同名。例如，假定类名是 BankAccount，则该类的构造函数必须命名为 BankAccount。

其二，构造函数定义不能返回值。此外，在函数声明起始处或函数头中，不允许指定返回类型(连 void 都不可以)。

例如，假定要添加构造函数来初始化图 10.5 的 BankAccount 类型的对象的余额及利率，可像下面这样定义类(省略了一些注释以节省篇幅，但其实应将它们包括在其中)：

```
class BankAccount
{
public:
    BankAccount(int dollars, int cents, double rate);
    // 将余额初始化为$dollars.cents，将利率初始化为百分之 rate

    void set(int dollars, int cents, double rate);
    void set(int dollars, double rate);
    void update();

    double getBalance();
    double getRate();
    void output(ostream& outs);
private:
    double balance;
    double interestRate;
    double fraction(double percent);
};
```

有三个地方需注意。其一，构造函数名称是 BankAccount，与类同名；其二，构造函数 BankAccount 的函数声明不以 void 或其他任何类型名称开头；其三，构造函数放在类定义的 public 小节。通常应将构造函数设为公共成员函数。所有构造函数都设为私有，将无法声明那个类的任何对象，这使类没有任何用处。

基于重新定义的 BankAccount，可以像下面这样声明并初始化 BankAccount 类型的两个对象：

```
BankAccount account1(10, 50, 2.0), account2(500, 0, 4.5);
```

假定构造函数的定义能执行我们承诺的初始化操作，上述语句就会声明对象 account1，将 account1.balance 的值设为 10.50，并将 account1.interestRate 的值设为 2.0。所以，

通过初始化对象 account1 获得了一个银行账户，其中含有余额 10.50 美元，利率为 2.0%。类似地，通过初始化对象 account2 获得了一个银行账户，其中含有余额 500.00 美元，利率为 4.5%。具体地说，就是先声明对象 account1，再调用构造函数 BankAccount，并向其传递 3 个参数：10，50 和 2.0。类似地，声明了 account2 之后，调用构造函数 BankAccount，并传递参数 500，0 和 4.5。结果在概念上等价于以下语句(虽然在 C++中不能以这种方式来写)：

```
BankAccount account1, account2; // 有问题，但可以解决
account1.BankAccount(10, 50, 2.0); // 完全不合法
account2.BankAccount(500, 0, 4.5); // 完全不合法
```

正如注释指出的那样，不可直接将上述 3 行代码放到程序中。第 1 行在修改后虽然可以接受，但接着对构造函数 BankAccount 的两个调用是非法的。不可像调用普通成员函数那样调用构造函数。但是，它们清楚地表明了我们的意图。像下面这样声明对象 account1 和 account2，这些意图就会自动得到满足：

```
BankAccount account1(10, 50, 2.0), account2(500, 0, 4.5);
```

构造函数的定义方式和其他任何成员函数一样。例如，假如对 BankAccount 类的定义进行修订，添加刚才描述的构造函数，那么还需要添加构造函数的定义：

```
BankAccount::BankAccount(int dollars, int cents, double rate)
{
    if ((dollars < 0) || (cents < 0) || ( rate < 0 ))
    {
        cout << "Illegal values for money or rate.\n";
        exit(1);
    }
    balance = dollars + 0.01 * cents;
    interestRate = rate;
}
```

由于类和构造函数同名，所以 BankAccount 这一名称会在函数头中出现两次：作用域解析操作符::之前的 BankAccount 是类名，之后的是构造函数名。还要注意，构造函数定义头中没有指定返回类型，就连 void 类型也没有指定。除了这些区别，构造函数的定义方式与普通成员函数无异。

可以重载构造函数名称(比如 BankAccount::BankAccount)，这和重载其他任何成员函数名称没有区别(比如图 10.5 中对 BankAccount::set 进行的重载)。事实上，构造函数通常都要重载，以便采取多种方式初始化对象。例如在图 10.6 中，我们重新定义了 BankAccount 类，提供了构造函数的三个版本。构造函数 BankAccount 的三个版本分别接收三个参数(和前面讨论的一样)、两个参数和 0 个参数。

图 10.6　具有构造函数的类

```
1  // 该程序用于演示 BankAccount 类
2  #include <iostream>
3  using namespace std;

4  // 定义银行账户的类:
5  class BankAccount
6  {
7  public:
8      BankAccount(int dollars, int cents, double rate);
9      // 将账户余额初始化为$dollars.cents，将利率初始化为百分之 rate

10     BankAccount(int dollars, double rate);
```

这个 BankAccount 定义是图 10.5 中的 BankAccount 类的增强版本

```
11      // 将账户余额初始化为$dollars.00，将利率初始化为百分之 rate
12      BankAccount();  ←———— 默认构造函数
13      // 将账户余额初始化为$0.00，将利率初始化为 0.0%
14      void set(int dollars, int cents, double rate);
15      // 后条件：将余额设为$dollars.cents，将利率设为百分之 rate
16
17      void set(int dollars, double rate);
18      // 后条件：将余额设为$dollars.00，将利率设为百分之 rate
19
20      void update();
21      // 后条件：将 1 年的单利添加到账户余额上
22      double getBalance();
23      // 返回当前账户余额
24      double getRate();
25      // 将当前账户利率作为一个百分比返回
26      void output(ostream& outs);
27      // 前条件：如果 outs 是一个文件输出流，那么 outs 已经连接到一个文件
28      // 后条件：账户余额和利率写入 outs 流
29  private:
30      double balance;
31      double interestRate;
32      double fraction(double percent);
33      // 将百分比转换成小数。例如，fraction(50.3)返回 0.503
34  };
35
36  int main()
37  {
38      BankAccount account1(100, 2.3), account2;  ←———— 这个声明将导致对默认构造函数的
                                                          调用。注意，这里没有使用圆括号
39      cout << "account1 initialized as follows:\n";
40      account1.output(cout);
41      cout << "account2 initialized as follows:\n";
42      account2.output(cout);
43      account1 = BankAccount(999, 99, 5.5);  ←———— 对构造函数 BankAccount::BankAccount
44      cout << "account1 reset to the following:\n";   的显式调用
45      account1.output(cout);
46      return 0;
47  }
48
49  BankAccount::BankAccount(int dollars, int cents, double rate)
50  {
51      if ((dollars < 0) || (cents < 0) || (rate < 0))
52      {
53          cout << "Illegal values for money or interest rate.\n";
54          exit(1);
55      }
56      balance = dollars + 0.01*cents;
57      interestRate = rate;
58  }
59
60  BankAccount::BankAccount(int dollars, double rate)
61  {
62      if ((dollars < 0) || (rate < 0))
63      {
64          cout << "Illegal values for money or interest rate.\n";
65          exit(1);
66      }                                          其他成员函数的定义与
67      balance = dollars;                         图 10.5 中的一样
68      interestRate = rate;
69  }
70
71  BankAccount::BankAccount(): balance(0), interestRate(0.0)
```

```
72  {
73     // 主体有意留空
74  }
```

屏幕输出

```
account1 initialized as follows:
Account balance $100.00
Interest rate 2.30%
account2 initialized as follows:
Account balance $0.00
Interest rate 0.00%
account1 reset to the following:
Account balance $999.99
Interest rate 5.50%
```

例如，声明 BankAccount 类型的一个对象时，假定只提供两个参数，如下所示：

```
BankAccount account1(100, 2.3);
```

那么对象 account1 在初始化之后，就代表一个余额为 100.00 美元，利率为 2.3%的账户。

另一方面，如果不提供任何参数，如下所示：

```
BankAccount account2;
```

那么对象在初始化之后，就代表一个余额为 0.00 美元，利率为 0.0%的账户。注意，假如构造函数没有参数，就不要在对象声明中包括任何圆括号。下面的语句是错误的：

```
BankAccount account2(); // 错误！不要这样做！
```

有一套完善的构造函数之后，某些情况下可以省略赋值成员函数(比如 set)。可用图 10.6 重载的 BankAccount 构造函数创建新的 BankAccount 对象，用你选择的值初始化它。然而，调用构造函数会创建新对象，所以如果只是想更改现有对象中的成员变量，可以考虑使用赋值函数。这正是本例没有删除两个 set 成员函数定义的原因。

构 造 函 数

构造函数是类的成员函数，名称和类相同。声明类的对象会自动调用构造函数。构造函数用于初始化对象。构造函数必须与定义它的类同名。

有必要详细讨论一下图 10.6 中的无参构造函数，因为其中出现了以前没有见过的一些东西。为便于参考，下面复制了这个无参构造函数的定义：

```
BankAccount::BankAccount() : balance(0), interestRate(0.0)
{
  // 主体有意留空
}
```

新元素在第一行，也就是以一个冒号开头的部分。构造函数定义的这个部分称为**初始化区域**。如上例所示，初始化区域在用于结束参数列表的那个圆括号之后，并在函数主体的起始花括号之前。这个初始化区域包括一个冒号，冒号后面是一个列表，列表中包括以逗号分隔的部分或全部成员变量。在每个成员变量名称之后，是放到一对圆括号中的初始值。上述构造函数定义完全等价于以下定义：

```
BankAccount::BankAccount()
{
```

```
        balance = 0;
        rate = 0.0;
    }
```

对于带有初始化区域的构造函数，其主体并非一定为空。例如，图 10.6 的那个接收两个参数的构造函数可改写成如下形式：

```
    BankAccount::BankAccount(int dollars, double rate)
                          : balance(dollars), interestRate(rate)
    {
        if ((dollars < 0) || (rate < 0))
        {
            cout << "Illegal values for money or interest rate.\n";
            exit(1);
        }
    }
```

注意，构造函数的参数可作为初始值提供。

构造函数初始化区域

如果有必要，可在构造函数初始化区域中初始化类的部分或全部成员变量。该区域在代表参数列表结束的那个圆括号之后，并在函数主体起始花括号之前。初始化区域的组成包括一个冒号，后跟以逗号分隔的部分或全部成员变量列表。每个成员变量后，都用一对圆括号来包含它的初始值。下例使用了构造函数初始化区域，它等价于图 10.6 中接收 3 个参数的那个构造函数。

示例

```
      BankAccount::BankAccount(int dollars, int cents, double rate)
      : balance(dollars + 0.01*cents), interestRate(rate)
{
    if ((dollars < 0) || (cents < 0) || (rate < 0))
    {
        cout <<
           "Illegal values for money or interest rate.\n";
        exit(1);
    }
}
```

注意，初始值可以是构造函数的参数。

任何时候声明类类型的对象都会自动调用构造函数，但声明后可再次调用。这样就可方便地设置对象的所有成员。下面澄清一下技术细节。调用构造函数会创建具有新值的匿名对象。所谓匿名对象，是指尚未由任何变量命名的对象。可将匿名对象赋给已命名对象(类变量)。例如，以下代码：

```
    account1 = BankAccount(999, 99, 5.5);
```

它调用构造函数 BankAccount 创建一个匿名对象，其余额为 999.99 美元，利率为 5.5%，再将该匿名对象赋给对象 account1(类类型的变量)使 account1 也代表一个余额为 999.99 美元，利率为 5.5%的账户。

可以看出，构造函数和普通函数的行为一样，只是它返回自己的类类型的一个对象。但由于调用构造函数总是创建新对象，而调用 set 成员函数只是更改现有成员变量的值，所以为了修改成员变量的值，调用 set 比调用构造函数更高效。要提高效率，或者在不创

建新对象的前提下更改成员变量的值，类定义中应同时准备 set 成员函数和构造函数。

调用构造函数

声明对象时会自动调用构造函数，但声明对象时必须提供构造函数要求的参数。也可显式调用构造函数创建类的新对象。

语法(声明一个对象，前提是有构造函数)

```
ClassName ObjectName(ArgumentsForConstructor);
```

示例

```
BankAccount account1(100, 2.3);
```

语法(显式调用构造函数)

```
Object = ConstructorName(ArgumentsForConstructor);
```

示例

```
account1 = BankAccount(200, 3.5);
```

构造函数必须与所属的类同名。所以在上述语法描述中，*ClassName* 和 *ConstructorName* 是同一个标识符。

如对象作为动态变量创建，还可指定初始化值，例如：

```
BankAccount *myAcct;
myAcct = new BankAccount (300,4.2)
```

编程提示：总是包括默认构造函数

C++并非总是为自定义类生成默认构造函数。如果你没有定义，编译器会生成一个什么事情都不做的默认构造函数。声明类的对象时会调用它。相反，如定义了至少一个构造函数，C++就不会生成其他构造函数。每次声明类的对象时，C++会查找最合适的构造函数定义。如声明对象时没有为构造函数提供参数，C++会查找默认构造函数；如果没有定义默认构造函数，C++会徒劳而返。

例如，假定像下面这样定义类：

```
class SampleClass
{
public:                                     获取两个实参的构造函数
    SampleClass(int parameter1, double parameter2);
    void do_stuff();
private:
    int data1;
    double data2;
};
```

为了声明 SampleClass 类的对象并调用构造函数，以下语句完全合法：

```
SampleClass myObject(7, 7.77);
```

但如果告诉你以下语句非法，你或许会感到惊讶：

```
SampleClass yourObject;
```

编译器将上述声明解释成要调用无参构造函数，但类定义中无此函数。要么必须在 yourObject 的声明中添加两个参数，要么必须定义一个无参构造函数。

可在无实参的情况下调用的构造函数称为**默认构造函数**。声明对象而不提供任何实参，就会默认调用它。由于经常都要在不提供构造函数实参的情况下声明对象，所以始终应该包括一个默认构造函数。下面是 SampleClass 的修订版本，其中包括一个默认构造函数：

```
class SampleClass
{
public:
     SampleClass(int parameter1, double parameter2);
     SampleClass();  ←—————— 默认构造函数
     void do_stuff();
private:
    int data1;
    double data2;
};
```

以这种方式重新定义了 SampleClass 类之后，前面的 yourObject 声明就合法了。

如果不希望默认构造函数初始化任何成员变量，只需在实现时提供一个空主体。下面的构造函数定义完全合法。调用时，除了让编译器不报错，它不会做其他任何事情：

```
SampleClass::SampleClass()
{
   // 什么都不做
}
```

使用 new 操作符创建类类型的动态变量会调用类的默认构造函数。■

陷阱：无参构造函数

假定 BankAccount 类的一个构造函数有两个形参，那么可以声明一个对象，并像下面这样将参数提供给构造函数：

```
BankAccount account1(100, 2.3);
```

为了调用无参构造函数，你也许会想当然地这样声明对象：

```
BankAccount account2(); // 这会导致问题
```

毕竟，从前在调用无参的函数时，包括的就是一对空的圆括号。但这对于构造函数来说是错误的。此外，这样做也许不会产生错误消息，因为它具有其他含义，只是这个含义也许不是你想要的。在编译器看来，上述代码是一个名为 account2 的函数的声明，该函数不接收任何参数，返回类型为 BankAccount 的值。

声明对象时，如希望 C++使用无参构造函数，千万不要添加圆括号。用无参构造函数来声明 account2，正确写法如下：

```
BankAccount account2;
```

然如果在赋值语句中显式调用构造函数，就必须使用圆括号。根据图 10.6 的定义和声明，以下语句会将 account1 的账户余额设为 0.00 美元，利率设为 0.0%：

```
account1 = BankAccount();
```

■

无参构造函数

声明对象时要调用无参构造函数就不要包括圆括号。例如，为了声明对象，并向构造函数传递两个参数，可以像下面这样写：

```
BankAccount account1(100, 2.3);
```

但是，要用无参构造函数声明对象，必须像下面这样写：

```
BankAccount account1;
```

不要像下面这样声明对象：

```
BankAccount account1(); // 不正确的声明
```

问题在于，上述语法实际会声明一个名为 account1 的函数，它不接收任何参数，返回一个 BankAccount 对象。

C++11 的成员初始化器和构造函数委托

视频讲解：*Default Initialization of Member Variables*

C++11 支持大多数面向对象编程语言现已支持的一个功能，即成员初始化，作用是为成员变量设置默认值。创建对象时，成员变量自动初始化为指定值。例如以下 Coordinate 类的定义和实现。

```
class Coordinate
{
public:
    Coordinate();
    Coordinate(int x);
    Coordinate(int x, int y);
    int getX();
    int getY();
private:
    int x=1;
    int y=2;
};
Coordinate::Coordinate()
{ }
Coordinate::Coordinate(int xVal) : x(xVal)
{ }
Coordinate::Coordinate(int xVal, int yVal) : x(xVal), y(yVal)
{ }
int Coordinate::getX()
{
    return x;
}
int Coordinate::getY()
{
    return y;
}
```

创建 Coordinate 对象，成员变量 x 和 y 分别设为默认值 1 和 2。调用构造函数显式设置可重写这些值。以下代码将 c1 的 x 和 y 设为默认值，但 c2 只将 y 设为默认值，x 通过输入的实参来显式设置。

```
Coordinate c1, c2(10);
cout << c1.getX() << " " << c1.getY() << endl; // Outputs 1 2
```

```
cout << c2.getX() << " " << c2.getY() << endl; // Outputs 10 2
```

C++11 支持的另一个相关功能是构造函数委托，即允许一个构造函数调用另一个。例如，可修改默认构造函数的实现，让它调用具有两个参数的构造函数。

```
Coordinate::Coordinate() : Coordinate(99,99)
{ }
```

执行 Coordinate c1;创建对象会调用默认构造函数，后者调用另一个构造函数将 x 设为 99，y 设为 99。

√ 自测题

22. 假定程序包含以下类定义(另外还有成员函数的定义)：

```
class YourClass
{
public:
    YourClass(int newInfo, char moreNewInfo);
    YourClass();
    void doSutff();
private:
    int information;
    char moreInformation;
};
```

以下哪些语句是合法的?

```
YourClass anObject(42, 'A');
YourClass anotherObject;
YourClass yetAnotherObject();
anObject = YourClass(99, 'B');
anObject = YourClass();
anObject = YourClass;
```

23. 怎样修改图 10.4 中的 DayOfYear 类定义，使其拥有两个(重载的)构造函数版本？一个版本有两个 int 形参(一个表示月份，一个表示天数)，它要设置私有成员变量来表示那个月份和天数。另一个版本没有形参，它应该将表示的日期设为 January 1(1 月 1 日)。修改时，两个构造函数中都不要使用构造函数初始化区域。

24. 重做自测题 23，用构造函数初始化区域初始化所有成员函数。

10.3 抽象数据类型

我们都知道——《泰晤士》报知道——但我们假装不知道。[①]

——*弗吉尼亚·伍尔芙[②](1882—1941)，《星期一或星期二》*

数据类型(比如 int 类型)有具体的、指定的值，比如 0，1，-1，2，等等。你或许以为值就是数据类型的全部。但同样重要的还有对这些值执行的运算(操作)。int 类型的运算

---

① 《星期一或星期二》是意识流代表作家伍尔芙的短篇小说集，其中包括 8 个故事。这句话出自其中第 4 个故事“一篇未完成的小说”。“《泰晤士》报”在这里代表类，“我们”代表程序员。类最大的优点就是隐藏实现细节，使我们可以“假装不知道”。——译注

② 弗吉尼亚·伍尔芙(Virginia Woolf)，英国女作家，被誉为二十世纪现代主义与女性主义的先锋。两次世界大战期间，她是伦敦文学界的核心人物，同时也是布卢姆茨伯里派(Bloomsbury Group)的成员之一。最知名的小说包括《戴洛维夫人》《灯塔行》《雅各的房间》《奥兰多》以及散文《属于自己的房间》等。——译注

涉及+，-，*，/，%以及其他几个操作符和预定义库函数。不要将数据类型简单地视为值的集合。**数据类型**由值的集合以及为那些值定义的一组基本运算构成。

使用数据类型的程序员访问不了值和运算的实现细节，该数据类型就称为**抽象数据类型**(Abstract Data Type，ADT)。预定义类型(比如 int)就是抽象数据类型(ADT)。对于 int 类型，你不知道+和-等运算具体如何实现。即使知道，也无法在任何 C++程序中利用这方面的信息。

程序员定义的类型(比如结构类型和类类型)不会自动成为 ADT。除非精心定义，并小心使用，否则程序员定义的类型在使用时可能显得非常“别扭”，造成程序难于理解和修改。为避免这些问题，最好的办法就是确保你定义的所有数据类型都是 ADT。C++要用类达此目的，但并非每个类都是 ADT。类要成为 ADT，必须采取特定方式定义，这正是下一节的主题。

用于生成抽象数据类型的类

类是你自己定义的一个类型，而不是那些已定义好的类型(比如 int 和 char)。对于类类型的一个值，它其实是成员变量值的一个集合。例如，图 10.6 的 BankAccount 类型的一个值包含 double 类型的两个数字。为便于参考，下面复制了类定义(只省略了注释)：

```
class BankAccount
{
public:
    BankAccount(int dollars, int cents, double rate);
    BankAccount(int dollars, double rate);
    BankAccount();
    void set(int dollars, int cents, double rate);
    void set(int dollars, double rate);
    void update();
    double getBalance();
    double getRate();
    void output(ostream& outs);
private:
    double balance;
    double interestRate;
    double fraction(double percent);
};
```

BankAccount 类型的用户不需要知道具体如何实现 BankAccount::update 以及其他任何成员函数。成员函数 BankAccount::update 目前的定义如下：

```
void BankAccount::update()
{
    balance = balance + fraction(interestRate)*balance;
}
```

可以摒弃私有函数 fraction，采用以下稍复杂的方式实现成员函数 update：

```
void BankAccount::update()
{
    balance = balance + (interestRate/100.0)*balance;
}
```

使用 BankAccount 类的程序员不必关心 update 具体如何实现，因两种实现的效果一样。

类似，使用 BankAccount 类的程序员不必关心类的值如何实现。我们选择将值实现为两个 double 值。如果 vacationSavings 是 BankAccount 类型的对象，那么 vacationSavings 的值由两个 double 值构成，这两个值分别存储在以下两个成员变量中：

```
vacationSavings.balance
vacationSavings.interestRate
```

但你不希望将 vacationSavings 对象的值看成是 double 类型的两个数，比如 1.3546e+2 和 4.5。相反，希望将 vacationSavings 的值看成是单一记录项：

```
Account balance $135.46
Interest rate 4.50%
```

这正是我们在 BankAccount::output 的实现中为什么要以这种格式来输出值的原因。

我们决定将这个 BankAccount 值作为两个 double 值(1.3546e+2 和 4.5)来实现，这是一种实现细节。完全可以采取其他方式实现这个 BankAccount 值，比如作为两个 int 值(135 和 46，分别表示余额中的美元数和美分数)和一个 double 值(0.045)来实现。值 0.045 就是 4.5%转换成小数之后的结果，它也许是实现百分比的一种更有用的方式。毕竟，为了计算一个账户的利息，我们最终都要将百分比转换成这样的小数。以这种方式实现 BankAccount 类，公共成员可保持不变，但私有成员要变成以下形式：

```
class BankAccount
{
public:
    <这一部分和以前完全一样>
private:
    int dollarsPart;  // 美元部分
    int centsPart;  // 美分部分
    double interestRate;
    double fraction(double percent);
};
```

与上述改动匹配需更改成员函数定义，但这并不复杂。例如，getBalance 的函数定义以及一个版本的构造函数可以更改为以下形式：

```
double BankAccount::getBalance()
{
    return (dollarsPart + 0.01 * centsPart);
}
BankAccount::BankAccount(int dollars, int cents, double rate)
{
    if ((dollars < 0) || (cents < 0) || ( rate < 0 ))
    {
        cout << "Illegal values for money or rate.\n";
        exit(1);
    }
    dollarsPart = dollars;
    centsPart = cents;
    interestRate = rate;
}
```

类似地，可重新定义其他每个成员函数，以适应新的账户余额和利率存储方式。

总之，虽然用户可将账户余额视为单个数，但那并不表示一定要在实现中采用单个

double 类型的数。如上例所示，它可以是 int 类型的两个数。使用 BankAccount 类型的程序员不必了解 BankAccount 类型的值具体如何实现。

这些围绕 BankAccount 类型进行的讨论演示了将类定义成抽象数据类型的基本思路。为了将类定义成抽象数据类型，需严格区分其他程序员使用类型的方式与类型的实现细节。这个区分要非常彻底，这样才能对类的实现进行自由更改。与此同时，使用 ADT 类的任何程序都不需要进行任何改动。达此目的需遵循以下规则。

- 所有成员变量都设为类的私有成员。
- 用户程序员需要的每个基本操作都设为类的公共成员函数，并完善地规定如何使用每个公共成员函数。
- 任何辅助函数都设为私有成员函数。

第 11 章和第 12 章将讲解定义 ADT 的其他方式，但以上规则是确保类成为抽象数据类型的常见方式。

ADT 的**接口**规定如何在程序中使用 ADT。定义 C++ ADT 类时，接口由类的公共成员函数以及对其用法进行描述的注释构成。要在程序中使用 ADT，该 ADT 的接口是你唯一要了解的。

ADT 的**实现**则指出该接口如何用 C++代码实现。ADT 的实现由类的私有成员以及公共/私有成员函数的定义构成。虽然需要这些实现才能运行一个使用 ADT 的程序，但在编写使用 ADT 的代码时，不需要了解关于实现的任何事情；换言之，不需要知道实现细节，就能写出程序的 main 部分，并写出 main 部分要用到的任何非成员函数。第 4 章和第 5 章曾针对普通函数的定义提出了类似的思路。与普通函数的实现相似，ADT 的实现也应存在于“黑盒”中，从外面看不见其中的东西。

视频讲解：*Separate Interface and Implementation*

第 12 章会讲述如何将 ADT 的接口与实现放到相互独立而且与使用 ADT 的程序分开的文件中。这样一来，使用 ADT 的程序员将完全看不到实现。但在那一章之前，我们准备将 ADT 类的所有实现细节放到与程序的 main 部分相同的文件中。虽然如此，仍然认为接口(类定义的 public 小节)和实现(类定义的 private 小节和成员函数定义)是 ADT 中完全独立的两个部分。我们要精心编写 ADT，使 ADT 的用户只需要知道 ADT 的接口，不需要知道和实现有关的任何事情。要验证是不是 ADT，只需验证在更改了 ADT 的实现之后，程序的其他部分是否不需要任何更改就能正常工作。下一个“编程实例”将对此进行演示。

ADT 最明显的好处就是在更改实现细节的同时，不需要更改程序的其他部分。但 ADT 还有另一些好处。将类变成 ADT，编程工作可由不同程序员分担。一个程序员负责设计和编写 ADT，其他程序员则负责使用 ADT。即使你是某个项目的唯一程序员，也可将较大的任务分解成两个较小的任务，以简化设计和调试。

编程实例　类的另一种实现

图 10.7 以另一种方式实现了 ADT 类 BankAccount。在这个版本中，银行账户的数据作为三个成员值来实现：一个存储账户余额的整数美元部分；一个存储账户余额的美分零头部分；还有一个存储利率。

图 10.7　BankAccount 类的另一个实现

```
1  // 该程序用于演示 BankAccount 类的另一个实现
2  #include <iostream>
3  #include <cmath>
4  using namespace std;
5  // 定义银行账户的类：
6  class BankAccount
7  {
8  public:
9      BankAccount(int dollars, int cents, double rate);
10     // 将账户余额初始化为$dollars.cents，将利率初始化为百分之 rate
11     BankAccount(int dollars, double rate);
12     // 将账户余额初始化为$dollars.00，将利率初始化为百分之 rate
13     BankAccount();
14     // 将账户余额初始化为$0.00，将利率初始化为 0.0%
15     void set(int dollars, int cents, double rate);
16     // 后条件：将余额设为$dollars.cents，将利率设为百分之 rate
17
18     void set(int dollars, double rate);
19     // 后条件：将余额设为$dollars.00，将利率设为百分之 rate
20
21     void update();
22     // 后条件：将 1 年的单利添加到账户余额上
23     double getBalance();
24     // 返回当前账户余额
25     double getRate();
26     // 将当前账户利率作为一个百分比返回
27     void output(ostream& outs);
28     // 前条件：如果 outs 是一个文件输出流，那么 outs 已经连接到一个文件
29     // 后条件：账户余额和利率写入 outs 流
30 private:
31     int dollarsPart;
32     int centsPart;
33     double interestRate; // 表示成小数，例如用 0.057 表示 5.7%
34     double fraction(double percent);
35     // 将百分比转换成小数，例如，fraction(50.3)返回 0.503
36     double percent(double fractionValue);
37     // 将小数转换成百分比，例如，percent(0.503)返回 50.3
38 };
39 int main()
40 {
41     BankAccount account1(100, 2.3), account2;
42
43     cout << "account1 initialized as follows:\n";
44     account1.output(cout);
45     cout << "account2 initialized as follows:\n";
46     account2.output(cout);
47
48     account1 = BankAccount(999, 99, 5.5);
49     cout << "account1 reset to the following:\n";
50     account1.output(cout);
51     return 0;
52 }
53
54 BankAccount::BankAccount(int dollars, int cents, double rate)
55 {
56     if ((dollars < 0) || (cents < 0) || (rate < 0))
57     {
58         cout << "Illegal values for money or interest rate.\n";
```

注意，BankAccount 的公共成员在外观和行为上与图 10.6 中的完全相同

新增（第 36 行）

由于 main 的主体和图 10.6 完全相同，所以屏幕输出也和图 10.6 完全相同（第 43 行）

```
59          exit(1);
60      }
61      dollarsPart = dollars;
62      centsPart = cents;
63      interestRate = fraction(rate);
64  }
65
66  BankAccount::BankAccount(int dollars, double rate)
67  {
68      if ((dollars < 0) || (rate < 0))
69      {
70          cout << "Illegal values for money or interest rate.\n";
71          exit(1);
72      }
73      dollarsPart = dollars;
74      centsPart = 0;
75      interestRate = fraction(rate);
76  }
77
78  BankAccount::BankAccount() : dollarsPart(0), centsPart(0), interestRate(0.0)
79  {
80      // 主体有意留空
81  }
82
83  double BankAccount::fraction(double percentValue)
84  {
85      return (percentValue/100.0);
86  }
87
88  // 使用 cmath:
89  void BankAccount::update()
90  {
91      double balance = getBalance();
92      balance = balance + interestRate*balance;
93      dollarsPart = staticCast<int>(floor(balance));
94      centsPart = staticCast<int>(floor((balance - dollarsPart)*100));
95  }
96
97  double BankAccount::getBalance()
98  {
99      return (dollarsPart + 0.01*centsPart);
100 }
101
102 double BankAccount::percent(double fractionValue)
103 {
104     return (fractionValue*100);
105 }
106
107 double BankAccount::getRate()
108 {
109     return percent(interestRate);
110 }
111
112 // 使用 iostream:
113 void BankAccount::output(ostream& outs)
114 {
115     outs.setf(ios::fixed);
116     outs.setf(ios::showpoint);
117     outs.precision(2);
118     outs << "Account balance $" << getBalance() << endl;
119     outs << "Interest rate " << getRate() << "%" << endl;
120 }
121
122 void BankAccount::set(int dollars, int cents, double rate)
123 {
124     if ((dollars < 0) || (cents < 0) || (rate < 0))
125     {
126         cout << "Illegal values for money or interest rate.\n";
127         exit(1);
```

在该 ADT 原来的实现中，私有成员函数 fraction 在 update 的定义中使用。在这个实现中，fraction改为在构造函数和set 函数的定义中使用

getBalance 和 getRate 的新的定义可确保输出仍然采用正确的单位

```
128     }
129     dollarsPart = dollars;
130     centsPart = cents;
131     interestRate = fraction(rate);
132 }

133 void BankAccount::set(int dollars, double rate)
134 {
135     if ((dollars < 0) || (rate < 0))
136     {
137         cout << "Illegal values for money or interest rate.\n";
138         exit(1);
139     }
140     dollarsPart = dollars;
141     interestRate = fraction(rate);
142 }
```

注意，图 10.6 和图 10.7 的实现虽然都有名为 interestRate 的成员变量，但它们存储的值稍有区别。假定账户以 4.7%的利率支付利息，那么在图 10.6 的实现中(它基本上与图 10.5 的实现相同)，interestRate 的值是 4.7。但在图 10.7 的实现中，interestRate 的值是 0.047。图 10.7 的实现将利率作为小数存储，而不是作为百分比数字。在新的实现中，最基本的区别在于一旦设置利率，就会立即调用 fraction 函数将利率转换成小数。所以，在新的实现中，私有成员函数 fraction 要在构造函数的定义中使用，但不需要在成员函数 update 的定义中使用，因为成员变量 interestRate 的值已被转换成小数。在原来的实现中(参见图 10.5 和图 10.6)，情况则刚好相反。在原来的实现中，私有成员函数 fraction 不是在构造函数的定义中使用，而是在 update 的定义中使用。

虽然更改了 BankAccount 类的私有成员，但没有在类定义的 public 小节更改任何东西。公共成员函数具有相同的函数声明，而且行为与图 10.6 老版本的 ADT 类完全相同。例如，虽然新的实现将 4.7%等百分数存为 0.047 这样的小数，但成员函数 getRate 仍会返回值 4.7，就像图 10.5 那个最老的实现一样。类似地，成员函数 getBalance 返回 double 类型的单个值(带小数点的账户余额)，这和图 10.5 最老的实现一样。但就实现细节来说，新版本已将余额作为 int 类型的两个成员变量来存储，而不是像老版本那样将它作为 double 类型的单独一个成员变量来存储。

公共和私有成员函数需区别对待。只要公共成员函数的声明（ADT 类的接口）保持不变，使用类的任何程序就不必进行任何改动。但可自由添加、删除或更改任何私有成员函数。本例新增了名为 percent 的私有函数，它的作用与 fraction 函数相反。fraction 函数将百分比转换成小数，而 percent 函数将小数转换回百分比。例如，fraction(4.7) 返回 0.047；而 percent(0.047) 返回 4.7。

信 息 隐 藏

第 3 章介绍函数时已提到信息隐藏的问题。从函数的角度出发，**信息隐藏**是指函数应该作为黑盒使用；换言之，使用函数的程序员不需要知道函数的任何实现细节。这意味着使用函数的程序员只需知道函数声明及其用法注释。在抽象数据类型的定义中，使用私有成员变量和私有成员函数，以另一种方式实现了信息隐藏——但这种方式更进一步，不仅函数进行了隐藏，数据值也进行了隐藏。

自测题

25. 定义 C++ ADT 类时，应使成员变量成为公共还是私有？成员函数应该公共还是私有？
26. 定义 C++ ADT 类时，哪些项目被视为 ADT 接口的一部分？哪些又被视为 ADT 实现的一部分？
27. 假定你的朋友按 10.3 节描述的方法定义一个 C++ ADT 类。你的任务是写程序来使用该 ADT。也就是说，必须编写程序的 main 部分以及要在 main 部分中使用的任何非成员函数。ADT 很长，需要在短时间内赶出这个程序。ADT 的哪些部分是你必须要看的，哪些部分可放心忽略？
28. 重写图 10.7 接收三个参数和两个参数的构造函数，使所有成员变量都在构造函数初始化区域中设置。

10.4 继　承

C++最强大的功能之一就是**派生类**的使用。**继承**其实只是派生类机制的另一种说法。一个类从另一个类派生时，除了继承后者的一切，还可添加自己的功能。例如，假设定义一个交通工具(Vehicle)类，它的成员变量记录了交通工具的轮数和最大乘客数。类还有取值和赋值函数。再假设定义一个汽车(Automobile)类，它有和交通工具类一样的成员变量和函数。但汽车类还增加了一些新东西，比如油箱容量和车牌号，以及一些新成员函数。不需要重复交通工具类的成员变量和函数。相反，利用 C++继承机制，让汽车类继承交通工具类的所有成员变量和函数。

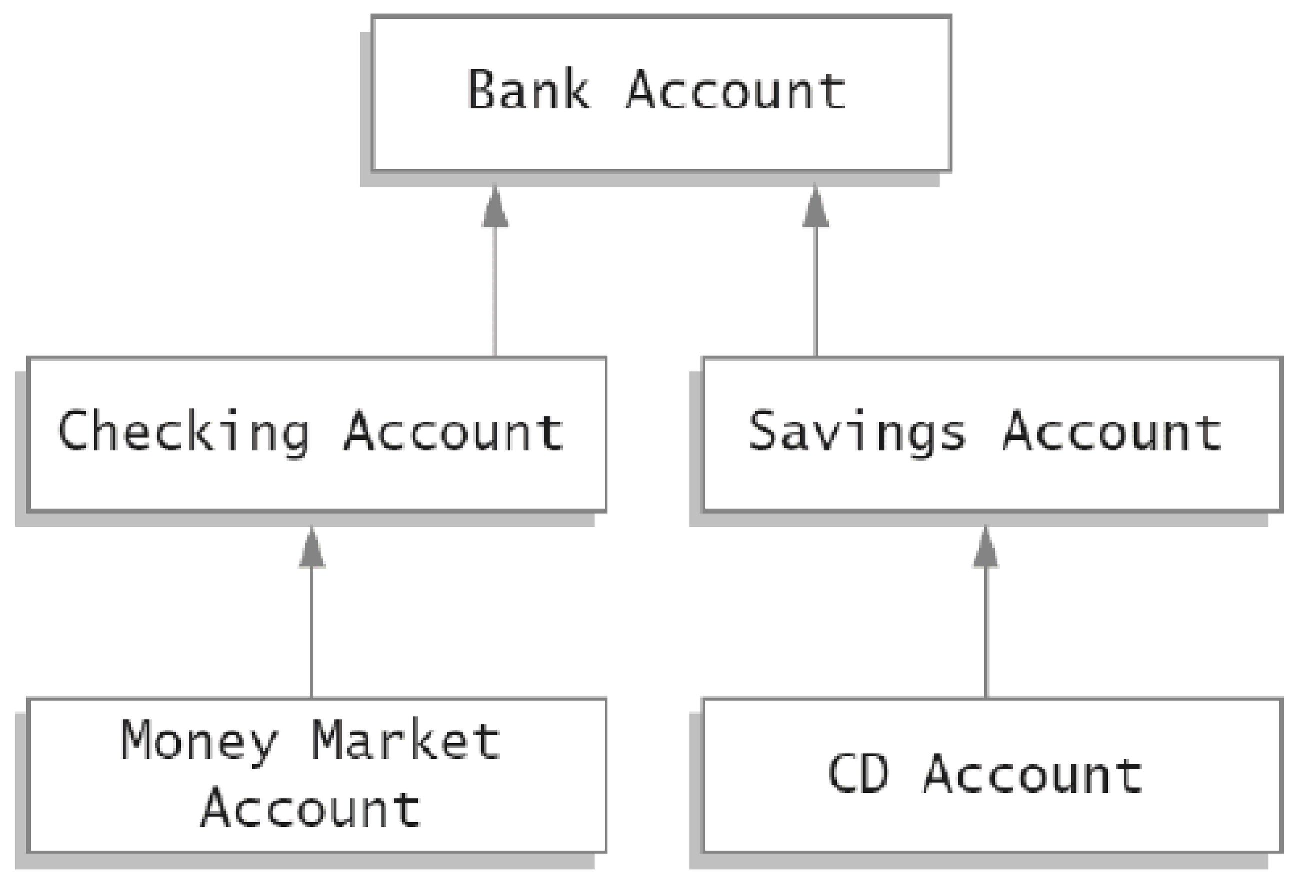
通过继承，我们可以先写比较常规的类，以后在其基础上添加新的细节，从而定义较具体的类。这减轻了工作量，因为较具体的类(派生类)继承了常规类的所有功能，程序员只需编码新功能。本节首先介绍继承和派生类的表示法，然后简要介绍怎样创建自己的派生类。继承的细节将留到第 15 章讲解。完全习惯派生类的思想需要一定时间，但很容易就能理解派生类的概念，并开始以简单和有用的方式使用它们。

派生类

以图 10.7 定义的 BankAccount 类为例。该类跟踪记录银行账户的金额和利率。这是相当常规的功能，适合任何计息账户。为了实现更具体的银行账户类型，可采用一个自然的层次结构对不同账户类型进行分组。图 10.8 展示了这个层次结构的一部分，其中包括银行账户(BankAccount)、支票账户(CheckingAccount)、货币市场账户(MoneyMarketAccount)、储蓄账户(SavingsAccount)以及定存账户(CDAccount)。

在层次结构中，BankAccount 是最常规的账户类型；它下方是较具体的账户类型。箭头从较具体账户类型指向较常规账户类型。除了表示不同的账户类型，每个框还对应一个 C++类。例如，支票账户能做银行账户能做的一切事情(存储金额和利率)，但还允许顾客存款和开支票。类似地，储蓄账户能做银行账户能做的一切事情，但还允许顾客存款和取款。和支票账户不同，储蓄账户可能不允许顾客开支票。由于支票和储蓄账户都是银行账户，所以在图 10.8 中，它们列在 BankAccount 类下方。说类 A 是类 B 的**派生类**时，是指类 A 具有类 B 的所有功能，同时新增了一些功能。根据约定，为了表示这个关系，要从较具体的类画一条箭头线指向较常规的类。例如在图 10.8 中，CheckingAccount 和 SavingsAccount 类就是 BankAccount 类的派生类。

图 10.8　类层次结构



在 C++中，类 A 可能是类 B 的派生类，而类 B 又可能是类 C 的派生类，依此类推。例如，定存账户和储蓄账户相似，区别是只有在“定存期”之后才能支取本金和利息。提前支取会有一定的惩罚(比如利息损失)。正是由于存在限制，所以定存账户的利息比一般储蓄账户高。为了表示它们的关系，在层次结构中，我们从 SavingsAccount 派生出 CDAccount。类似地，货币市场账户是特殊类型的支票账户，一般会限制顾客开的支票数量，规定的最低余额会高一些，不过会支付较高的利息，目的是吸引大额存款。在层次结构中，我们从 CheckingAccount 派生出 MoneyMarketAccount。

派生类经常借助继承和家族关系来讨论。如果类 B 是类 A 的派生类，则类 B 是类 A 的**子**，类 A 是类 B 的**父**。父类也称为**基类**。派生类**继承**了父类的成员函数。例如，每辆敞篷车都从汽车那里继承了“有 4 个轮子”的事实。这正是为什么派生类的主题经常牵涉到“继承”的原因。

定义派生类

为了创建类来表示储蓄账户，可以拷贝 BankAccount 类并重命名为 SavingsAccount，再添加新的公共成员函数来存款和取款。虽然可行，但效率太差，因为 SavingsAccount 类会重复 BankAccount 类的大多数功能。不仅浪费内存，还会造成以后难以修改。例如，假如以后决定修改 update()函数，每天而不是每年计息，那么 SavingsAccount 和 BankAccount 类都要修改。将 SavingsAccount 类定义成 BankAccount 类的派生类，则可有效解决这些问题。这样 SavingsAccount 类就可以共享 BankAccount 类定义好的变量和函数。要在定义派生类时指定这种关系，需添加一个冒号，后跟关键字 public 以及父类或基类名称。

```
class SavingsAccount : public BankAccount
{                     ↑———— 冒号将派生类 SavingsAccount 与其
public:                     父类 BankAccount 分隔开
   SavingsAccount(int dollars, int cents, double rate);
   <其他构造函数一般放在这里>
   void deposit(int dollars, int cents);    // 存款
   void withdraw(int dollars, int cents);   // 取款
private:
};
```

注意，我们只定义了储蓄账户特有的函数和数据，本例就是用于存取款的函数。不需要重新定义与银行账户有关的所有变量和函数，比如利率、美元、美分或 update()函数，那些成员将从 BankAccount 类继承，并在构造 SavingsAccount 对象时自动创建。例如，为了创建 SavingsAccount 对象，可以像下面这样调用函数：

```
SavingsAccount account(100, 50, 5.5);
account.deposit(10,25);
account.output(cout);
```

调用派生类 SavingsAccount 定义的函数

调用父类 BankAccount 定义的函数

在这个例子中，继承允许在派生类的上下文中重用父类定义的代码。此外，如果以后更改了 BankAccount 的某个函数——比如 update()——只要重新编译和链接程序，新代码就能自动在派生类的上下文中发挥作用。图 10.9 给出了 SavingsAccount 类的一个实现，并给出了用于测试存款和取款的 main 函数。为简化问题，存款和取款函数没有提供校验功能。例如，不检查负的金额。但是，用一些 if 语句可以轻松添加它们。

图 10.9 SavingsAccount 派生类

```
<在这里插入图 10.6 除 main 函数之外的一切。>
1 class SavingsAccount : public BankAccount
2 {
3 public:
4     SavingsAccount(int dollars, int cents, double rate);
5     // 其他构造函数放在这里
6     void deposit(int dollars, int cents);
7     // 在账户余额上加$dollars.cents
8     void withdraw(int dollars, int cents);
9     // 从账户余额中减$dollars.cents
10 private:
11 };
12 int main()
13 {
14     SavingsAccount account(100, 50, 5.5);
15     account.output(cout);
16     cout << endl;
17     cout << "Depositing $10.25." << endl;
18     account.deposit(10,25);
19     account.output(cout);
20     cout << endl;
21     cout << "Withdrawing $11.80." << endl;
22     account.withdraw(11,80);
23     account.output(cout);
24     cout << endl;
25     return 0;
26 }
27 SavingsAccount::SavingsAccount(int dollars, int cents, double rate):
28                               BankAccount(dollars, cents, rate)
29 {
30     // 主体有意留空
31 }
32 void SavingsAccount::deposit(int dollars, int cents)
33 {
34     double balance = getBalance();
35     balance += dollars;
36     balance += (static_cast<double>(cents) / 100);
37     int newDollars = static_cast<int>(balance);
38     int newCents = static_cast<int>((balance - newDollars) * 100);
39     set(newDollars, newCents, getRate());
40 }
```

冒号表明 SavingsAccount 类派生自 BankAccount 类

只需定义新成员函数和变量

SavingsAccount 构造函数调用 BankAccount 构造函数。注意冒号

deposit 函数在余额上加存款金额，通过 set 函数更改成员变量

```
41 void SavingsAccount::withdraw(int dollars, int cents)
42 {
43     double balance = getBalance();
44     balance -= dollars;
45     balance -= (static_cast<double>(cents) / 100);
46     int newDollars = static_cast<int>(balance);
47     int newCents = static_cast<int>((balance - newDollars) * 100);
48     set(newDollars, newCents, getRate());
49 }
```

withdraw 函数从余额中减取款金额，通过 set 函数更改成员变量

屏幕输出

```
Account balance $100.50
Interest rate 5.50%
Depositing $10.25.
Account balance $110.75
Interest rate 5.50%
Withdrawing $11.80.
Account balance $98.95
Interest rate 5.50%
```

定义好 SavingsAccount 类之后，就可以从 SavingsAccount 类派生出更具体的类。例如，为了定义定存账户类，需添加一个新的私有成员变量来存储距离到期日的天数，并定义函数来访问该变量：

```
class CDAccount : public SavingsAccount
{
public:
    CDAccount(int dollars, int cents, double rate,
              int daysToMaturity);
    <其他构造函数一般放在这里>
    int getDaysToMaturity();
    // 返回距离到期日的天数
    void decrementDaysToMaturity();
    // daysToMaturity 变量递减 1
private:
    int daysToMaturity; // 距离到期日的天数
};
```

同样只定义了定存账户特有的函数和数据。这个例子需存储和处理距离到期日的天数。和银行账户/储蓄账户有关的所有变量和函数都不需要重新定义，它们自动从父类继承。例如，一旦实现了 CDAccount 类特有的函数，就可创建 CDAccount 的对象，并为其调用来自 CDAccount，SavingsAccount 以及 BankAccount 类的下列函数：

```
// 新建定存账户，余额是$1000，利率是 6%，距离到期日的天数是 180 天
CDAccountnewCD(1000, 0, 6.0, 180);

newCD.deposit(100,50);   ← 调用 SavingsAccount 中的函数
daysToMaturity = newCD.getDaysToMaturity();   ← 调用 CDAccount 中的函数
// 返回 180
balance = newCD.getBalance();   ← 调用 BankAccount 中的函数
// 返回 1100.50
```

用继承能做许多事情，这个简单的例子只揭示了冰山之一角。第 15 章会描述更多细节。虽然要付出一番努力，才能真正用好继承并有效地设计类，但从长期看这些付出是值得的。目标是用继承写出更精简、更容易理解和维护的代码。

√ 自测题

29. 继承怎样支持代码重用，并使代码更容易维护？

30. 派生类能不能用父类私有成员变量的名称来直接访问该变量？

31. 假定 SportsCar 是 Automobile 的派生类，而且 Automobile 类有公共成员函数 accelerate 和 addGas。SportsCar 类的对象有名为 accelerate 和 addGas 的成员函数吗？

小　结

- 结构将不同类型的数据合并成一个(复合)数据值。
- 类将数据和函数合并成一个(复合)对象。
- 类的成员变量或成员函数既可以是公共的，也可以是私有的。如果是公共的，可以在类的外部使用；如果是私有的，只能在那个类的另一个成员函数的定义中使用。
- 函数形参可为类或结构类型。函数可返回类或结构类型的值。
- 类的成员函数可采取和普通函数一样的方式重载。
- 构造函数是类的成员函数，在声明该类的对象时自动调用。构造函数必须与定义它的类同名。
- 数据类型由一个值的集合以及为这些值定义的一组基本运算构成。
- 如果使用数据类型的程序员不需要知道那个类型的值和运算是如何实现的，就说该类型是抽象数据类型(ADT)。
- 在 C++中实现抽象数据类型，一个办法是将类的所有成员变量都定义成私有，将运算作为公共成员函数来实现。
- 继承是指类和类之间的父/子关系。子类或派生类继承父类的成员。

自测题答案

1. a. double
   b. double
   c. 非法——不能用结构类型名称取代结构变量名称
   d. 非法——未声明 savingsAccount 结构变量
   e. char
   f. TermAccount

2.
```
A $9.99
A $1.11
```

3. 许多编译器提供的错误消息都显得模棱两可。但令人惊讶的是，g++提供了非常准确的错误消息：

```
g++ -fsyntax-only c10testq3.cc
probl.cc:8: semicolon missing after declaration of 'Stuff'  // 'Stuff'的声明后缺少分号
probl.cc:8: extraneous 'int' ignored                        // 多余的'int'被忽略
probl.cc:8: semicolon missing after declaration of 'struct Stuff'
                                                 // 'struct Stuff'的声明后缺少分号
```

4. `A x = {1, 2};`

5. a. 虽然提供的初始值较少，但并非语法错误。初始化之后，dueDate.month == 12，dueDate.day == 21，dueDate.year == 0。没有提供初始值的成员变量被自动初始化为相应类型的“零”值。
   b. 正确初始化：12 == dueDate.month，21 == dueDate.day，2022 == dueDate.year。
   c. 错误：初始值太多。
   d. 也许是设计错误。代码作者只为日期的初始值提供了 2 位数。年份应该用 4 位数表示，使用 2 位日期的程序存在 Y2K 问题。

6.
```
struct EmployeeRecord
{
    double wageRate;
    int vacation;
    char status;
};
```

7.
```
void readShoeRecord(ShoeType& newShoe)
{
    cout << "Enter shoe style (one letter): ";
    cin >> newShoe.style;
    cout << "Enter shoe price $";
    cin >> newShoe.price;
}
```

8.
```
ShoeType discount(ShoeType oldRecord)
{
    ShoeType temp;
    temp.style = oldRecord.style;
    temp.price = 0.90*oldRecord.price;
    return temp;
}
```

9.
```
struct StockRecord
{
    ShoeType shoeInfo;
    Date arrivalDate;
};
```

10.
```
StockRecord aRecord;
aRecord.arrivalDate.year = 2006;
```

11.
```
void DayOfYear::input()
{
```

```
    cout << "Enter month as a number: ";
    cin >> month;
    cout << "Enter the day of the month: ";
    cin >> day;
}
```

12.
```
void Temperature::set(double newDegrees, char newScale)
{
    degrees = newDegrees;
    scale = newScale;
}
```

13. 圆点操作符和作用域解析操作符都用于成员名称，指出该成员是哪个类或结构的成员。像图 10.3 那样定义 DayOfYear 类，而且 today 是 DayOfYear 类的一个成员，可用圆点操作符访问 month 成员，即 today.month。给出成员函数的定义时，要用作用域解析操作符向编译器指出该函数是哪个类定义的(也就是位于作用域解析操作符::之前的那个类名)。

14.
```
void DayOfYear::checkDate()
{
   if ((month < 1) || (month > 12)
      || (day < 1) || (day > 31))
   {
      cout << "Illegal date. Aborting program.\n";
      exit(1);
   }

   if (((month == 4) || (month == 6) || (month == 9) || (month == 11)) && (day == 31))
   {
      cout << "Illegal date. Aborting program.\n";
      exit(1);
   }

   if ((month == 2) && (day > 29))
   {
      cout << "Illegal date. Aborting program.\n";
      exit(1);
   }
}
```

15.
```
hyundai.price = 4999.99; // 非法，price 是私有的
jaguar.setPrice(30000.97); // 合法
double aPrice, aProfit;// 合法
aPrice = jaguar.getPrice();// 合法
aProfit = jaguar.getProfit();// 非法，getProfit 是私有的
aProfit = hyundai.getProfit();// 非法，getProfit 是私有的
if (hyundai == jaguar) // 非法。==不能用于类
    cout << "Want to swap cars?";
hyundai = jaguar;// 合法
```

16. 修改后，除了下面这个语句，其他语句都是合法的：

```
if (hyundai == jaguar) // 非法，== 不能用于类
   cout << "Want to swap cars?";
```

17. private 变量仅限同一个类的成员函数访问。只有类作者提供的函数才能更改 private 变量。这样一来，类作者就能控制对这些私有数据的更改，防止不小心破坏类数据。

18. a. 只需要有一个。如果类(或结构)没有 public:成员，编译器会报错。
b. 不要求，可以一个都没有；但类一般至少要有一个 private:小节。
c. 在类中，这些成员默认为 private 成员。
d. 在结构中，这些成员默认为 public 成员。

19. 以下答案仅供参考：

```
double difference(BankAccount account1, BankAccount account2)
{
```

```
    return (account1.getBalance() - account2.getBalance());
}
```

注意，以下答案是错误的，因为 balance 是私有成员：

```
double difference(BankAccount account1, BankAccount account2)
{
    return (account1.balance - account2.balance); // 非法
}
```

20.
```
void doubleUpdate(BankAccount& theAccount)
{
    theAccount.update();
    theAccount.update();
}
```

由于不是成员函数，调用 update 时必须给出对象名和圆点操作符。

21.
```
BankAccount newAccount(BankAccount oldAccount)
{
    BankAccount temp;
    temp.set(0, oldAccount.getRate());
    return temp;
}
```

22.
```
YourClass anObject(42, 'A'); // 合法
YourClass anotherObject; // 合法
YourClass yetAnotherObject(); // 有疑问
anObject = YourClass(99, 'B'); // 合法
anObject = YourClass(); // 合法
anObject = YourClass; // 非法
```

标记为“// 有疑问”的语句严格说并不是非法的，只是它实际的意思也许并不是你希望的那样。如果你认为它是对名为 yetAnotherObject 的一个对象的声明，就是错误的。它实际是一个 yetAnotherObject 函数声明，该函数无参数，返回 YourClass 类型的一个值。理论上合法，但根据你的实际来说却是非法的。为了声明名为 yetAnotherObject 的对象，并用默认构造函数初始化它，以下语句才是正确的：

```
YourClass yetAnotherObject;
```

23. 修改过的类定义如下：

```
class DayOfYear
{
public:
    DayOfYear(int theMonth, int theDay);
    // 前条件：theMonth 和 theDay 构成一个可能的日期。根据参数来初始化日期

    DayOfYear();
    // 将日期初始化为 1 月 1 日

    void input();
    void output();
    int getMonth();
    // 返回月份，1 代表 1 月，2 代表 2 月，依此类推
    int getDay();
    // 返回当月天数
private:
    void checkDate();
    int month;
    int day;
};
```

注意，这里省略了成员函数 set，因为有了构造函数，就没必要使用 set。还必须添加以下函数定义(并删除 DayOfYear::set 的函数定义)：

```
DayOfYear::DayOfYear(int theMonth, int theDay)
```

```
{
    month = theMonth;
    day = theDay;
    checkDate();
}

DayOfYear::DayOfYear()
{
    month = 1;
    day = 1;
}
```

24. 类定义和上题相同。构造函数定义需要如下修改：

```
DayOfYear::DayOfYear(int theMonth, int theDay)
    : month(theMonth), day(theDay)
{
    checkDate();
}
DayOfYear::DayOfYear() : month(1), day(1)
{
    // 主体有意留空
}
```

25. 成员变量应该全部私有。用于构成 ADT 接口的成员函数(也就是代表 ADT 支持的各种运算的成员函数)则应该公共。仅在其他成员函数的定义中使用的辅助函数应该是私有的。

26. 所有私有成员变量声明都是实现的一部分(公共成员变量应该是不允许存在的)。类的公共成员函数的函数声明(在类的定义中列出)以及对这些函数声明进行描述的注释是接口的一部分。所有私有成员函数的函数声明都是实现的一部分。所有成员函数(不管公共还是私有成员函数)的定义都是实现的一部分。

27. 你只需看接口部分。换言之，只需要看类的公共成员的函数声明(它们在类定义中列出)以及对这些函数声明进行描述的注释。不需要看任何私有成员函数的函数声明、私有成员变量的声明、公共成员函数的定义或者私有成员函数的定义。

28.

```
BankAccount::BankAccount(int dollars, int cents,  double rate) :
                         dollarsPart(dollars),
                         centsPart(cents), interestRate(fraction(rate))
{
    if ((dollars < 0) || (cents < 0) || (rate < 0))
   {
        cout << "Illegal values for money or interest rate.\n";
        exit(1);
    }
}

BankAccount::BankAccount(int dollars, double rate) :
        dollarsPart(dollars), centsPart(0), interestRate(fraction(rate))
{
    if ((dollars < 0) || (rate < 0))
    {
        cout << "Illegal values for money or interest rate.\n";
        exit(1);
    }
}
```

29. 父类定义的函数和数据可在派生类中使用，所以程序员不必在派生类中重复定义。这增强了可维护性，因为相同的代码不必在多个类中重复。要修改代码，修改一处地方就可以了。此外，继承很好地隔离了派生类专有的代码。这些专有代码只在派生类的定义中出现，读起来会更容易一些。

30. 不能，但派生类能通过公共函数间接访问父类的私有成员变量。

31. 有，派生类能访问相同的函数。第 15 章将讨论如何使函数为两个类的对象做不同的事情。

编 程 练 习

编程练习一般只需要写短小的程序，要运用本章提到的编程概念。

1. 视频讲解：*Solution to Practice Program 10.1*

   重新定义图 10.1 的 CDAccount，把它变成类而不是结构。使用和图 10.1 一样的成员变量，但把它们变成私有的。为以下每一项操作都包括相应的成员函数：一个返回初始余额；一个返回到期时的余额；一个返回利率；另一个返回存期。写一个构造函数将所有成员变量设为任何指定的值，同时包括一个默认构造函数。将类定义嵌入一个测试程序。

2. 重新定义编程练习 1 的 CDAccount 类，新版本使用相同的接口，但实现不同。新的实现在许多方面都类似于 BankAccount 类的第二个实现(参见图 10.7)。在你实现的 CDAccount 类中，要将余额记录为两个 int 类型的值：一个用于美元；一个用于美分。代表利率的成员变量将利率作为小数而非百分比来存储。例如，利率 4.3%存储为 double 值 0.043。存期(term)的存储方式与图 10.1 一样。

3. 定义 CounterType 类，其对象用于计数，即记录一个非负整数。写默认构造函数将计数器设为零；写另一个构造函数获取一个参数，将计数器设为由参数指定的值。写成员函数使计数递增和递减 1。确保任何成员函数都不会将计数器设为负值。再写两个成员函数，一个返回当前计数值，另一个将计数值输出到一个流。用于输出的成员函数有一个 ostream 类型的形参，用于指定接收输出的输出流。将类定义嵌入测试程序。

编 程 项 目

编程项目要求综合运用多方面的知识来解决问题，程序一般比编程练习大，解题方式多样化。

1. 为一门课写评分程序，规则如下。
   a. 有两次随堂考，每次满分 10 分。
   b. 有一次期中考试和一次期末考试，每次满分 100 分。
   c. 期末考试成绩占总成绩的 50%，期中考试占 25%，两次随堂考总共占 25%(不要忘记对随堂考的分数进行正规化。应在求平均值之前将它们换算为百分数)。

   90 分和 90 分以上的成绩评为 A，80 分和 80 分以上的成绩(但低于 90 分)评为 B，70 分和 70 分以上的成绩(但低于 80 分)评为 C，60 分和 60 分以上的成绩(但低于 70 分)评为 D，所有低于 60 分的成绩评为 F。

   程序读取学生的分数，输出学生的考试记录。记录包括两次随堂考、一次期中考试和一次期末考试的成绩，另外还有学生的平均分(数值)以及最终的字母评分。为学生记录定义并使用一个结构。如果是课堂作业，询问输入/输出是用键盘/屏幕完成，还是用文件完成。如果使用文件，询问文件名。

2. 重做编程项目 1。这次用类而不是结构定义学生的考试记录。在类中为编程项目 1 提到的所有输入数据包含对应的成员变量。此外，用一个成员变量存储学生整学期的加权平均分(数值)，用另一个成员变量存储学生最终的字母评分。使所有成员变量都成为私有成员。添加一组成员函数，它们分别能够：将每个成员变量设为传给函数的实参；获取每个成员变量的当前值；一个 void 函数，它能计算学生的加权平均分(数值)并设置对应的成员变量；另一个 void 函数，它能计算学生最终的字母成绩并设置对应的成员变量。

3. 将 Month 类定义成代表月份的抽象数据类型。该类用 int 成员变量表示一个月(1 表示 1 月，2 表示 2 月，依此类推)。请包括以下所有成员函数：一个构造函数使用英文月份名称的前 3 个字母来设置月份名称，这 3 个字母要通过 3 个参数来接收；一个构造函数只接收一个 int 类型的参数，并用这个整数来设置月份(1 代表 1 月，2 代表 2 月，依此类推)；一个默认构造函数；一个输入函数，它将月份作为整数来读取；一个输入函数，它将月份作为月份名称的前 3 个字母来读取；一个输出函数，它将月份作为一个整数来输出；一个输出函数，它将月份作为月份名称的前 3 个字母来输出；以及一个成

员函数，它将下一个月作为 Month 类型的值返回。输入和输出函数各自有一个流参数。将类定义嵌入测试程序。

4. 重新定义编程项目 3 所描述的 Month 类的实现。这次将月份作为 char 类型的 3 个成员变量来实现；这些成员变量分别存储了月份名称的前 3 个字母。将定义嵌入测试程序。

5. (做这个项目前，必须先完成项目 3 或项目 4)。改写图 10.4 的程序，针对用于存储月份的成员变量，请将项目 3 或项目 4 定义的 Month 类用作它的类型。重新定义成员函数 output，使用一个 ostream 类型的参数来获取输出流。修改程序，使输出到屏幕的一切都同时输出到一个文件。这意味着所有输出语句都要执行两次；一次使用 cout 作为实参，另一次使用输出流作为实参。如果这是课堂作业，请向老师询问文件名。输入仍通过键盘进行，只是输出要发送到一个文件。

6. 我母亲去商店买东西时总是随身携带一个小的红色计数器。当她在商店里买完所有该买的东西之后，这个计数器就能显示出她本次购物总共花了多少钱。这个小东西有一个 4 数字显示屏，每个数字都提供了一个递增按钮。另外还提供一个重置按钮。如果总金额超过 99.99 美元，一个溢出指示灯就会变红(这已经是陈年旧事了)。

   编写和实现 Counter 类的成员函数，模拟并常规化这种古老的便携式计数器的行为。构造函数应创建一个 Counter 对象，该对象最多能计数到构造函数参数指定的一个数。也就是说，Counter(9999) 将生成最多能计数到 9999 的计数器。所有新构造的计数器都应该显示读数 0。成员函数 void reset();能在任何时候将计数器的数字重置为 0。成员函数 void incr1();使个位递增 1，void incr10();使十位递增 1，void incr100();和 void incr1000();则分别使百位和千位递增 1。每次递增，只需在私有数据成员上加一个恰当的数，不需要采取其他任何行动。成员函数 bool overflow();检测溢出(如果计数器的私有数据成员在递增之后，结果值超过了该计数器在构造时指定的最大值，就会发生溢出)。

   用这个类模拟前面描述的小型红色计数器。虽然显示的是一个整数，但最右边的两位数要被视为美分零头，最左边的两位数应被视为美元整数。

   分别为分、角、元和十元设计按键。遗憾的是，键盘上没有任何键是特别好记的。一个方案是选择 asdfo 这 5 个字母键：a 代表分，s 代表角，d 代表元，f 代表十元。按下每个字母键后，紧接着输入 1～9 的某个数字，最后按 Enter 键，使相应的数字递增。每次运算之后，都报告可能产生的溢出。要想直接请求溢出，可以按 O 键。

7. 写一个有理数类。第 11 章还会重做这道题；届时，操作符重载将极大简化编码。目前准备使用成员函数 add，sub，mul，div 和 less，它们分别执行+，-，*，/和<运算。例如，a + b 要写成 a.add(b)；a < b 要写成 a.less(b)。

   为有理数定义一个类。有理数是有限的数，表示成两个相除的整数。但除法不会真正执行，它只用于标识，比如 1/2，2/3，15/32，65/4，16/5。你应该将有理数表示成 2 个 int 值，分别称为 numerator(分子)和 denominator(分母)。

   抽象数据类型的一个原则是：必须为其定义构造函数，以便创建含有任何合法值的对象。你提供的构造函数应该根据提供的 int 值对来构造对象；该构造函数获取 2 个 int 参数。由于所有 int 值同时也是有理数(比如 2/1 或者 17/1)，所以还应提供接收单个 int 参数的构造函数。

   提供成员函数 input 和 output，它们分别获取一个 istream 参数和一个 ostream 参数，并采取 2/3 或者 37/51 这样的形式，从键盘或文件获取有理数，并将有理数写到屏幕或文件。

   提供成员函数 add(加)，sub(减)，mul(乘)和 div(除)来返回有理数。提供 less(小于)函数返回 bool 值。提供成员函数 neg，它没有参数，返回调用对象的相反数。提供 main 函数，全面测试你所实现的类。定义函数时，可参考以下公式：

```
a/b + c/d = (a*d + b*c) / (b*d)
a/b - c/d = (a*d - b*c) / (b*d)
(a/b) * (c/d) = (a*c) / (b*d)
(a/b) / (c/d) = (a*d) / (c*b)
-(a/b) = (-a/b)
(a/b) < (c/d) means (a*d) < (c*b)
(a/b) == (c/d) means (a*d) == (c*b)
```

所有正负号都放在分子部分，分母保持正数。

8. 定义 Odometer(里程表)类来跟踪汽车的油耗和里程。在其中包含 private 成员变量来跟踪行驶的英里数以及汽车的油耗 (单位是每加仑行驶英里数，即 MPG)。这个类应该有一个构造函数将这些值初始化为零。包含一个成员函数，将里程表重置为 0 英里；包含一个成员函数来设置油耗；包含一个成员函数来接受一趟旅行所行驶的英里数，把它加到里程表的总数里面；包含一个成员函数来返回自从里程表上一次重置以来，汽车消耗了多少加仑的燃油。

   在测试程序中使用该类，创建几趟虚拟的旅行，每次都指定不同的油耗。

9. 重做(或者第一次做)第 5 章的编程项目 7，但用一个类来封装日期。用私有成员变量存储日、月和年，并用恰当的构造函数和成员函数来获取/设置数据。创建一个 public 函数返回一个日期是星期几。所有辅助函数都应当声明为 private。将类定义嵌入合适的测试程序中。

10. 美国邮政局在每个信封上都打印条形码来表示一个 5 位(或更多位)的邮政编码。条形码使用一种称为 POSTNET 的格式(该格式已在 2009 年被新的 OneCode 系统取代)。条形码由长长短短的条构成，放大了看就像下面这样：

   本程序将条形码表示成由数字构成的字符串。1 表示长条，0 代表短条。所以，上述条形码在程序中像下面这样表示：

   110100101000101011000010011

   条形码第一位和最后一位始终是 1。把它们删除，就剩下 25 位。这 25 个数位划分为 5 位一组，就得到以下结果：

   10100　　10100　　01010　　11000　　01001

   再来考虑每一组 5 个数位。在每一组数位中，肯定不多不少有两个 1。这 5 个数位的每一个都对应一个加权值，分别是 7，4，2，1 和 0。将加权值和数位现有的值相乘，得到新的 5 个数位。新的 5 个数位的值相加，结果即为邮编的对应数位。下表演示了 10100 的编码。

条码数位	1	0	1	0	0
加权值	7	4	2	1	0
条码数位*加权值	7	0	2	0	0

   最终的邮编数位 ＝7＋0＋2＋0＋0＝9。
   如果上述结果等于 11，就表示对应的邮编数位是 0；这是必须的，因为每一组都有两个 1，所以不可能直接表示 0。

   为每一组 5 个数位都重复上述过程，连接起来得到完整的邮编。上面的示例条形码在完成解码后，得到最终的邮编是 99504。虽然 POSTNET 方案看起来似乎过于复杂，但采用这个设计之后，如果在扫描条形码的时候出错，电脑就可以检测到。

   写一个邮政编码(ZipCode)类来编码和解码美国邮政局在信封上使用的 5 位条形码。这个类有两个构造函数。第一个构造函数的参数是一个整数邮编(int zip)，比如 99504。第二个构造函数的参数是一个由 0 和 1 构成的条形码字符串(string code)，比如"110100..."。虽然内部有两种方式输入邮编，但类只应该使用一种格式来存储邮编(可自行选择把它存储为一个条码字符串，或者存储为一个整数邮编)。这个类还至少要有两个 public 成员函数，一个将邮编作为整数返回，另一个将条码格式的邮编作为一个字符串返回。所有辅助函数都声明为 private。将类定义嵌入一个合适的测试程序中。如果向构造函数传递了一个无效的条形码，程序应该打印一条错误消息。

11. 设计包含电影信息的 Movie 类。类中包含以下内容：
   - 电影名称

- MPAA 分级(例如 G，PG，PG-13，R)
- 打分为 1 的人数(Terrible，真难看)
- 打分为 2 的人数(Bad，不好看)
- 打分为 3 的人数(OK，还可以)
- 打分为 4 的人数(Good，好看)
- 打分为 5 的人数(Great，出色)

实现这个类，为电影名称和 MPAA 分级提供取值和赋值函数。写 addRating 函数获取整数作为参数，校验参数是不是在 1~5 之间。如果是，就递增和参数对应的打分人数。例如，输入 3 会造成打 3 分的人数递增 1。写 getAverage 函数返回电影的平均分。最后，添加构造函数允许程序员创建具有指定电影名和 MPAA 分级的对象。在构造函数中，将为电影打分的人数设为 0。写 main 函数测试类，创建至少两个电影对象，为每部电影添加至少 5 个打分，输出电影名、MPAA 分级以及平均分。

第 11 章　类中的友元函数、重载操作符和数组

有工具，事竟成。

——*温斯顿·丘吉尔(1874—1965)，1941 年 2 月 9 日的广播演讲*

概述

本章讲解更多为类定义函数和操作符的技术，包括对常用操作符(比如+，*和/)进行重载，使它们像应用于预定义类型 int 和 double 那样应用于你定义的类。

预备知识

本章基于第 2 章～第 10 章的知识。

11.1　友元函数

信任你的朋友。

——*俗语*

以前是将类的操作(比如输入、输出和取值函数等)作为类的成员函数来实现。但对于某些操作，更自然的做法是作为普通(非成员)函数来实现。本节讨论如何将应用于对象的操作定义成非成员函数。先研究一个简单例子。

编程实例　一个相等性函数

第 10 章开发了名为 DayOfYear 的类，它记录一个日期(比如 January 1 或 July 4)，这个日期可能是节假日、生日或其他纪念日。我们不断改进这个类，最后的版本是在第 10 章的自测题 23 中生成的。图 11.1 重复了 DayOfYear 的最终版本，并再次改进，添加了一个 equal 函数，测试 DayOfYear 类型的两个对象，判断它们的值是否表示同一个日期。

图 11.1　相等性函数

```
1   // 该程序用于演示 equal 函数。DayOfYear 类与第 6 章自测题 23～24 中相同
2   #include <iostream>
3   using namespace std;

4   class DayOfYear
5   {
6   public:
7       DayOfYear(int theMonth, int theDay);
8       // 前条件：theMonth 和 theDay 构成一个可能的日期。根据参数来初始化日期

9       DayOfYear();
10      // 将日期初始化为 1 月 1 日

11      void input();

12      void output();

13      int getMonth();
14      // 返回月份，1 代表 1 月，2 代表 2 月，依此类推

15      int getDay();
16      // 返回当月天数
```

```
17  private:
18      void checkDate();
19      int month;
20      int day;
21  };
22
23  bool equal(DayOfYear date1, DayOfYear date2);
24  // 前条件：date1 和 date2 已被赋值
25  // 如果 date1 和 date2 表示相同的日期，就返回 true，否则返回 false
26
27  int main()
28  {
29      DayOfYear today, bachBirthday(3, 21);
30      cout << "Enter today's date:\n";
31      today.input();
32      cout << "Today's date is ";
33      today.output();
34      cout << "J. S. Bach's birthday is ";
35      bachBirthday.output();
36      if ( equal(today, bachBirthday) )
37          cout << "Happy Birthday Johann Sebastian!\n";
38      else
39          cout << "Happy Unbirthday Johann Sebastian!\n";
40      return 0;
41  }
42
43  bool equal(DayOfYear date1, DayOfYear date2)
44  {
45       return ( date1.getMonth() == date2.getMonth() &&
46                date1.getDay() == date2.getDay() );
47  }
48
49  DayOfYear::DayOfYear(int theMonth, int theDay)
50                  : month(theMonth), day(theDay)
51  {
52      checkDate();
53  }
54
55  int DayOfYear::getMonth()
56  {
57      return month;
58  }
59
60  int DayOfYear::getDay()
61  {
62      return day;
63  }
64
65  // 使用 iostream:
66  void DayOfYear::input()
67  {
68      cout << "Enter the month as a number: ";
69      cin >> month;
70      cout << "Enter the day of the month: ";
71      cin >> day;
72  }
74
75  // 使用 iostream:
76  void DayOfYear::output()
77  {
78      cout << "month = " << month
79           << ", day = " << day << endl;
80  }
```

注意，equal 不是 DayOfYear 的成员函数。它是在类的外部定义的

省略的函数和构造函数定义和第 10 章的自测题 14 和自测题 24 一样，那些细节目前不需要

示范对话

```
Enter today's date:
Enter the month as a number: 3
Enter the day of the month: 21
Today's date is month = 3, day = 21
J. S. Bach's birthday is month = 3, day = 21
Happy Birthday Johann Sebastian!
```

假定 today 和 bachBirthday 是 DayOfYear 类型的两个对象，而且都被赋了日期值。为了测试它们是否代表相同的日期，可使用以下布尔表达式：

```
equal(today, bachBirthday)
```

如果 today 和 bachBirthday 代表同一日期，equal 函数调用返回 true。在图 11.1 中，这个布尔表达式用于控制一个 if-else 语句。

equal 函数的定义很直观。如两个日期代表同一个月，以及一个月中的同一天，两个日期就相等。equal 的定义使用取值函数 getMonth 和 getDay 来比较两个对象代表的月份和天数。

注意，没有将 equal 函数作为成员函数。虽然可以将 equal 作为 DayOfYear 的成员函数，但要记住，equal 比较 DayOfYear 的两个对象。如果将 equal 变成成员函数，就必须确定调用对象是第一个日期，还是第二个日期。我们不准备在两个日期中任意挑选一个作为调用对象，而是以相同方式对待两个日期。我们使 equal 成为一个普通(非成员)函数，它获取两个日期作为参数。■

√ 自测题

1. 为名为 before 的函数写定义，函数获取 DayOfYear 类型(在图 11.1 中定义)的两个参数，返回一个布尔值。如第一个参数代表的日期在第二个参数代表的日期之前，就返回 true；否则返回 false。例如，1 月 5 日就在 2 月 2 日之前。

友元函数

如果类有一套完整的取值函数，可利用取值函数定义一个普通函数来测试相等性，或执行其他任何需要依赖私有成员变量的计算。这样虽能访问私有成员变量，但效率堪忧。重新分析图 11.1 的 equal 函数定义。读取月份必须调用取值函数 getMonth，读取天数必须调用取值函数 getDay。虽然可行，但假如能直接访问成员变量，代码会更简单、更高效。

对于图 11.1 的 equal 函数，一个更简单、更有效的定义如下：

```
bool equal(DayOfYear date1, DayOfYear date2)
{
    return ( date1.month == date2.month && date1.day == date2.day );
}
```

上述定义只有一个问题：它是非法的！之所以非法，是因为成员变量 month 和 day 是 DayOfYear 类的私有成员。私有成员变量(和私有成员函数)通常不能在一个函数的主体中引用，除非函数是一个成员函数。现在的问题是，equal 并非 DayOfYear 的成员函数。但有一个办法可以为非成员函数赋予和成员函数一样的访问权限。让 equal 函数成为 DayOfYear 的友元，上述 equal 定义就完全合法了。

类的**友元函数**不是这个类的成员函数，而是一个“友好”的函数，它能像成员函数那

样访问类的私有成员。友元函数可直接读取成员变量的值，甚至能直接更改成员变量的值。换言之，在赋值语句中，可将私有成员变量放在赋值操作符的任何一侧。为了使函数成为友元函数，必须在类定义中把它命名为友元。图 11.2 修改了 DayOfYear 类的定义，使 equal 函数成为该类的友元。在类定义中声明友元函数只需在函数声明前添加关键字 friend。

为了将友元函数添加到类定义中，需要列出它的函数声明，这和列出成员函数的声明没什么区别，只是必须在友元函数声明之前添加关键字 friend。注意，友元不是成员函数，它本质上仍然是普通函数，只是被特别授予了访问类的数据成员的权限。友元的定义和调用方式与普通函数无异。特别注意，图 11.2 给出的 equal 函数定义没有在函数头中包括限定符 DayOfYear::。此外，调用 equal 函数时不使用圆点操作符。equal 函数获取 DayOfYear 类型的对象作为参数，这一点也和其他非成员函数没有区别。但在友元函数的定义中，可直接访问类的私有成员变量和私有成员函数(使用它们的名称)。也就是说，它具有和成员函数相同的访问权限。

图 11.2　相等性函数作为友元函数

```
1   // 演示 equal 函数
2   // 在这个版本中，equal 成为 DayOfYear 类的友元
3   #include <iostream>
4   using namespace std;
5
6   class DayOfYear
7   {
8   public:
9       friend bool equal(DayOfYear date1, DayOfYear date2);
10      // 前条件：date1 和 date2 已被赋值
11      // 如果 date1 和 date2 表示相同的日期，就返回 true，否则返回 false
12      DayOfYear(int theMonth, int theDay);
13      // 前条件：theMonth 和 theDay 构成一个可能的日期。根据参数来初始化日期
14      DayOfYear();
15      // 将日期初始化为 1 月 1 日
16      void input();
17      void output();
18      int getMonth();
19      // 返回月份，1 代表 1 月，2 代表 2 月，依此类推
20      int getDay();
21      // 返回当月天数
22  private:
23      void checkDate();
24      int month;
25      int day;
26  };
27
28  int main()
29  {
           <程序的 main 部分与图 11.1 相同>
30  }
31
32  bool equal(DayOfYear date1, DayOfYear date2)
33  {
34      return ( date1.month == date2.month &&
35               date1.day == date2.day );
36  }
```

注意，可通过名称访问私有成员变量 month 和 day

本程序剩余部分，包括“示范对话”，和图 11.1 一样

编程提示：定义取值函数和友元函数

从表面看，使所有基本函数都成为类的友元，就没必要在类中包括取值和赋值函数。毕竟，友元函数能访问私有成员变量，取值和赋值函数似乎多余。这种说法不完全错误。如果真的使世界上所有函数都成为类的友元，确实不需要取值和赋值函数。但是，让所有函数都成为友元不现实。

为了理解为什么仍然需要取值函数，让我们深入分析图 11.2 给出的 DayOfYear 类。假定要在另一个程序中使用该类，而那个程序可能想对一个 DayOfYear 对象的“月份”部分采取一些操作。例如，可能想计算一年还剩多少月。具体地说，程序的 main 部分可能包含以下代码：

```
DayOfYear today;
cout << "enter today's date: \n";
today.input();
cout << "There are " << (12 - today.getMonth())
     << " months left in this year.\n";
```

不能将 today.getMonth()替换成 today.month，因为 month 是类的私有成员。这时需要使用取值函数 getMonth。

这个例子说明，绝对有必要在自己的类中包括取值函数。另一些情况下，则可能需要使用赋值函数。你也许觉得，由于经常需要用的是取值和赋值函数，所以用不着友元。这在某种程度上是正确的。注意，既可将 equal 函数定义成友元，不使用取值函数(图 11.2)；也可不定义成友元，而使用取值函数(图 11.1)。大多数时候，将函数变成友元的唯一理由就是简化函数定义，并提高效率。但这些理由通常就足够了。■

友 元 函 数

类的友元函数本质上是普通函数，只是能访问那个类的对象的私有成员。为了使函数成为类的友元，必须在类定义中为友元函数列出函数声明。函数声明前要附加关键字 friend。函数声明可放在 private 或者 public 区域。但无论如何它都是公共函数，所以较好的做法是在 public 区域列出它。

语法(含有友元函数的类定义)

```
class ClassName
{
public:
    friend DeclarationForFriendFunction_1  <------ 不需要首先列出友元函数，可将
    friend DeclarationForFriendFunction_2          它们夹杂在这些函数声明中间
        ...
    MemberFunctionDeclarations
private:
    PrivateMemberDeclarations
};
```

示例

```
class FuelTank // 油箱
{
public:
```

```
        friend double needToFill(FuelTank tank);
        // 前条件：tank 的成员变量必须赋值，返回加满油箱所需的汽油升数
        FuelTank(double theCapacity, double theLevel);
        FuelTank();
        void input();
        void output();
    private:
        double capacity; // 油箱容积，以升为单位
        double level;  // 剩余油量
    };
```

友元函数不是成员函数。友元函数的定义和调用方式与普通函数相同。调用友元函数时不使用圆点操作符。友元函数的定义也不使用类型限定符。

编程提示：同时使用成员函数和非成员函数

成员函数和友元函数作用很相似。事实上，有时甚至不好决定应该将函数作为类的友元，还是作为类的成员函数。大多数时候，一个函数既可作为成员函数，也可作为友元函数，并能以相同方式执行相同任务。但某些情况确实更适合使用成员函数，而另一些情况更适合使用友元函数(有时甚至更适合使用普通的、老式的函数，比如图 11.1 那个版本的 equal)。以下两个简单的规则可帮你选择成员函数和非成员函数。

- 函数要执行的任务只涉及一个对象，就使用成员函数。
- 要执行的任务涉及多个对象，就使用非成员函数。例如，图 11.1(和图 11.2)的 equal 函数涉及两个对象，所以将它作为非成员(友元)函数。

无论将非成员函数作为友元函数，还是使用取值和赋值函数，归根结底都取决于效率和个人习惯。只要有足够多的取值和赋值函数，两种方式都合适。

但决定使用成员还是非成员函数时，不能完全依赖上述两个简单的规则。随着经验的增长，你会发现，某些时候需要违反这些规则。一个更准确、但更难理解的规则是：如果任务与单个对象密切相关，就使用成员函数；如果任务涉及多个对象，而且对象被均匀地使用，就使用非成员函数。但在刚开始的时候，这个更准确的规则很难捉摸。除非你能游刃有余地处理对象，否则坚持按前面两个简单的规则行事。■

编程实例　Money 类(版本 1)

图 11.3 定义 Money 类来表示美元金额。值作为单个整数值实现，单位是分。例如，9.95 美元作为值 995 存储。由于用整数表示金额，所以金额能准确表示。没有使用只能表示近似值的 double 类型，我们当然希望货币金额完全准确。

图 11.3　Money 类(版本 1)

```
1  // 该程序用于演示 Money 类
2  #include <iostream>
3  #include <cstdlib>
4  #include <cctype>
5  using namespace std;
6  // 表示美元金额的类
7  class Money
8  {
9  public:
```

```
10      friend Money add(Money amount1, Money amount2);
11      // 前条件：amount1 和 amount2 已被赋值
12      // 返回 amount1 和 amount2 的值之和

13      friend bool equal(Money amount1, Money amount2);
14      // 前条件：amount1 和 amount2 已被赋值
15      // 如果 amount1 和 amount2 具有相同的值，就返回 true；否则返回 false

16      Money(long dollars, int cents);
17      // 初始化对象，根据参数中的美元和美分数额来表示一个金额
18      // 如果金额是负数，那么美元和美分都必须为负

19      Money(long dollars);
20      // 初始化对象，使它的值表示$dollars.00

21      Money();
22      // 初始化对象，使它的值表示$0.00

23      double getValue();
24      // 前条件：调用对象已被赋值
25      // 返回在调用对象的数据中记录的金额

26      void input(istream& ins);
27      // 前条件：如果 ins 是一个文件输入流，那么 ins 已经连接到一个文件
28      // 在输入流 ins 中，已输入了一个金额，其中包括美元符号
29      // 负数金额的表示方式为-$100.00
30      // 后条件：调用对象的值被设为来自输入流 ins 的金额
31      void output(ostream& outs);
32      // 前条件：如果 outs 是一个文件输出流，那么 outs 已经连接到一个文件
33      // 后条件：调用对象中记录好的美元符和金额将发送到输出流 outs
34  private:
35      long allCents;
36  };

37  int digitToInt(char c);
38  // Money::input 的定义中要用到的函数的函数声明：
39  // 前条件：c 是'0'～'9'的一个数字
40  // 返回与该数字对应的整数；例如，digitToInt('3')返回 3

41  int main()
42  {
43      Money yourAmount, myAmount(10, 9), ourAmount;
44      cout << "Enter an amount of money: ";
45      yourAmount.input(cin);
46      cout << "Your amount is ";
47      yourAmount.output(cout);
48      cout << endl;
49      cout << "My amount is ";
50      myAmount.output(cout);
51      cout << endl;

52      if (equal(yourAmount, myAmount))
53          cout << "We have the same amounts.\n";
54      else
55          cout << "One of us is richer.\n";
56      ourAmount = add(yourAmount, myAmount);
57      yourAmount.output(cout);
58      cout << " + ";
59      myAmount.output(cout);
60      cout << " equals ";
61      ourAmount.output(cout);
62      cout << endl;
63      return 0;
64  }
65  Money add(Money amount1, Money amount2)
66  {
67      Money temp;
68
69      temp.allCents = amount1.allCents + amount2.allCents;
70      return temp;
71  }
72
```

```
73  bool equal(Money amount1, Money amount2)
74  {
75      return (amount1.allCents == amount2.allCents);
76  }
77
78  Money::Money(long dollars, int cents)
79  {
80      if (dollars * cents < 0) // 如果一个为负，一个为正
81      {
82          cout << "Illegal values for dollars and cents.\n";
83          exit(1);
84      }
85      allCents = dollars*100 + cents;
86  }
87
88  Money::Money(long dollars) : allCents(dollars * 100)
89  {
90      // 主体有意留空
91  }
92
93  Money::Money() : allCents(0)
94  {
95      // 主体有意留空
96  }
97
98  double Money::getValue()
99  {
100     return (allCents * 0.01);
101 }
102 // 使用 iostream，cctype，cstdlib:
103 void Money::input(istream& ins)
104 {
105     char oneChar, decimalPoint,
106          digit1, digit2; // 代表美分数额的数字
107     long dollars;
108     int cents;
109     bool negative; // 如果输入负数，就设为 true
110
111     ins >> oneChar;
112     if (oneChar == '-')
113     {
114         negative = true;
115         ins >> oneChar; // 读入'$'
116     }
117     else
118         negative = false;
119     // 如果输入合法，那么 oneChar == '$'
120
121     ins >> dollars >> decimalPoint >> digit1 >> digit2;
122
123     if ( oneChar != '$' || decimalPoint != '.'
124          || !isdigit(digit1) || !isdigit(digit2) )
125     {
126         cout << "Error illegal form for money input\n";
127         exit(1);
128     }
129     cents = digitToInt(digit1) * 10 + digitToInt(digit2);
130
131     allCents = dollars * 100 + cents;
132     if (negative)
133         allCents = -allCents;
134 }
135
136 // 使用 cstdlib 和 iostream:
137 void Money::output(ostream& outs)
138 {
139     long positiveCents, dollars, cents;
140     positiveCents = labs(allCents);
141     dollars = positiveCents/100;
142     cents = positiveCents%100;
143
144     if (allCents < 0)
```

```
145         outs << "-$" << dollars << '.';
146     else
147         outs << "$" << dollars << '.';
148
149     if (cents < 10)
150         outs << '0';
151     outs << cents;
152 }
153
154 int digitToInt(char c)
155 {
156     return (static_cast<int>(c) - static_cast<int>('0'));
158 }
```

示范对话

```
Enter an amount of money: $123.45
Your amount is $123.45
My amount is $10.09
One of us is richer.
$123.45 + $10.09 equals $133.54
```

整数金额(以分为单位)存储在名为 allCents 的变量中。可将成员变量 allCents 设为 int 类型。但对于某些编译器，能表示的最大金额可能受到严重限制。C++的某些实现只用两个字节存储 int 类型。①用两个字节实现，最大 int 值只比 32 000 大一点点。而 32 000 分只能表示 320 美元，这是相当小的金额。由于希望处理比 320 美元多得多的金额，所以选择将成员变量 allCents 的类型设为 long。那些用两个字节来实现 int 类型的 C++编译器通常用 4 个字节实现 long 类型。long 类型可表示 int 类型能表示的所有值。但由于用 4 个字节实现，所以 long 类型允许的最大值要比 int 类型的最大值大得多。在大多数系统上，long 类型的最大值都在 20 亿以上(long 类型也称为 long int。long 和 long int 这两个名称表示同一种数据类型)。

Money 类有两个操作被定义成友元函数：equal 和 add(在图 11.3 定义)。add 函数返回一个 Money 对象，它的值是两个实参的值之和。如 amount1 和 amount2 这两个对象的值代表相同的金额，equal(amount1, amount2)返回 true。

注意，Money 类根据我们平时的习惯读写金额，比如$9.95 或者-$9.95。首先分析成员函数 input(也在图 11.3 中定义)。该函数首先读取一个字符，它要么为美元符号('$')，要么为负号('-')。如第一个字符是负号，函数就会记住金额为一个负数，方法是将变量 negative 的值设为 true。接着，它读取下一个字符，这应该是美元符号。另一方面，如果第一个字符不是'-'，negative 会设为 false。此时已读取了负号(如果有的话)和美元符号。随后，input 函数将整数美元数额作为 long 类型的值读入，并将这个美元数额存储到名为 dollars 的局部变量中。读了美元部分之后，input 函数将剩余的部分作为 char 类型的值读入；它会读取 3 个字符，它们应该是小数点和两个数位。

定义成员函数 input 时，你或许想将小数点作为 char 类型的值读入，然后将美分数额作为 int 类型的值读入。但我们不打算这样做，因为某些 C++编译器以特殊方式对待前置的零。正如 11.1 节所解释的那样，许多编译器在读取具有前置零的数字时，都不会按照你希望的方式进行。例如$7.09 这个金额，如果 C++代码将 09 作为 int 类型的值读入，

① 参考第 2 章，图 2.2 总结了大多数现代编译器所实现的数值类型。

就可能产生不正确的结果。

以下赋值语句将构成美分部分的两个数位转换成整数，并将这个整数存储到局部变量 cents 中：

```
cents = digitToInt(digit1) * 10 + digitToInt(digit2);
```

执行这个赋值语句后，cents 的值就是输入金额中的美分数额。

辅助函数 digitToInt 获取一个数位(比如'3')作为实参，将其转换成相应的 int 值(3)。之所以需要这个辅助函数，是因为成员函数 input 将美分数额中的两个数位当作两个 char 值读取，两个值分别存储在局部变量 digit1 和 digit2 中。但一旦数字读入计算机，就希望它们能作为真正的数字使用。所以，要用 digitToInt 函数将'3'这样的数位转换成数字(比如 3)。digitToInt 函数的定义在图 11.3 中给出。你只需无条件地相信这个定义能够做我们假定它能做的事情，也就是将整个函数视为一个“黑盒”。目前只需知道 digitToInt('0')返回 0，digitToInt('1')返回 1，依此类推。然而，要想知道这个函数具体如何工作，也不是太难。下个选读小节解释了 digitToInt 的具体实现。

将局部变量 dollars 和 cents 分别设为输入金额中的美元数和美分数之后，成员变量 allCents 就很容易设置了。以下赋值语句将 allCents 设为正确的美分总数：

```
allCents = dollars * 100 + cents;
```

然而，这样总是将 allCents 设为正数金额。如金额为负，allCents 的值就必须从正数变成负数。这是用以下语句来实现的：

```
if (negative)
    allCents = -allCents;
```

成员函数 output(在图 11.3 中定义)根据成员变量 allCents 的值计算美元和美分数额。这是通过整除 100 来实现的。例如，假定 allCents 的值是 995(美分)，美元数额就是 995/100，结果是 9；而美分数额是 995%100，结果是 95。所以，在 allCents 为 995(美分)的前提下，$9.95 是最终输出的值。

在成员函数 output 的定义中，允许输出负数形式的金额。负数的整除结果没有标准定义，不同的 C++实现可能有所不同。为了避免这个问题，在执行除法运算前先获取 allCents 的绝对值。为了计算绝对值，我们使用预定义函数 labs。labs 函数返回与其实参对应的绝对值，这和 abs 函数一样。然而，labs 获取的是 long 类型的实参，并返回 long 值。labs 函数在头文件为 cstdlib 的库中(和 abs 函数同一个库)。注意，有的 C++版本没有包括 labs。如果你的 C++实现不包括 labs 函数，自己定义一个也不难。

实现 digitToInt(选读)

下面摘录了图 11.3 的 digitToInt 函数定义：

```
int digitToInt(char c)
{
    return (static_cast<int>(c) - static_cast<int>('0'));
}
```

返回值的公式表面上有点儿奇怪，但细节并不复杂。参数 c 是要转换的数位(比如'3')，返回值是对应的 int 值(比如 3)。正如第 2 章和第 6 章讲到的那样，char 类型的值作为数字实现。遗憾的是，实现'3'所用的数字并不是 3。通过强制类型转换 static_cast<int>(c)，可获得用于实现字符 c 的数字，该数字会转换成 int 类型。这样一来，c 就从 char 类型转换成了 int 类型。但这个数字并不是我们希望的数字。例如，int('3')的结果并不是 3，而是其他一个数。需要将 static_cast<int>(c)转换成与 c 对应的数字(例如，将'3'转换成 3)。所以必须对 static_cast<int>(c)的结果进行修正，以获得真正想要的数字。

我们知道数字是按顺序排列的。所以，int('0') + 1 等于 int('1')，int('1') + 1 等于 int('2')，int('2') + 1 等于 int('3')，依此类推。利用这个特点，就能让 digitToInt 返回正确的值。如果 c 是'0'，那么返回值如下：

```
static_cast<int>(c) - static_cast<int>('0')
```

也就是：

```
static_cast<int>('0') - static_cast<int>('0')
```

所以 digitToInt('0')返回 0。再看 c 值为'1'时的情况。返回值是：

```
static_cast<int>(c) - static_cast<int>('0')
```

也就是：

```
static_cast<int>(1) - static_cast<int>('0')
```

这相当于：

```
(static_cast<int>('0') + 1) - static_cast<int>('0')
```

也就是：

```
static_cast<int> ('0') - static_cast<int>('0') + 1
```

由于 static_cast<int>('0') - static_cast<int>('0')等于 0，所以结果是 0 + 1。还可检查其他数位('2'～'9')；每个数位生成的数字都比上一个大 1。

陷阱：数字常量中的前置零

下面是图 11.3 的 main 部分给出的对象声明：

```
Money yourAmount, myAmount(10, 9), ourAmount;
```

myAmount(10, 9)中的两个实参代表$10.09。由于经常用“.09”这样的格式写美分数，所以可能习惯性地将对象声明写成 myAmount(10, 09)。但这会造成问题。在数学中，9 和 09 表示同一个数。但有的 C++编译器根据前置的零来判断是否属于某个不同种类的数字。所以在 C++中，常量 9 和 09 并不一定表示同一个数字。对于某些编译器，前置零表示数字采用八进制，而不是十进制。由于八进制没有 9，所以常量 09 在 C++中没有意义。常量 00～07 应该能正确地工作，因为它们在八进制和十进制中具有相同的含义。但某些特殊系统即使处理 00～07 也会遇到麻烦。

ANSI C++标准要求输入默认解释为十进制，忽略前置零。GNU C++编译器(g++)和

Microsoft 的 VC++编译器都符合这一标准，所以它们在处理前置零时不会有什么问题。大多数编译器厂商都开始遵循 ANSI 标准，所以应该是与 ANSI C++标准相容的，这个前置零的问题最终有望得到完全解决。建议写一个小程序，在自己的编译器上测试这一点。 ■

√ 自测题

2. 一个类的友元函数和成员函数有哪些相同点，又有哪些不同点？
3. 假定要为图 11.2 定义的 DayOfYear 类添加友元函数。友元函数名为 after，接收 DayOfYear 类型的两个实参。如果第一个实参代表的日期在第二个实参代表的日期之后，函数返回 true，否则返回 false。例如，2 月 2 日在 1 月 5 日之后。在图 11.2 的 DayOfYear 类的定义中，需添加哪些内容？
4. 假定要为图 11.3 定义的 Money 类添加友元函数(名为 subtract)来执行减法运算。在图 11.3 的 Money 类的定义中，需添加哪些内容？subtract 函数获取两个 Money 类型的实参，返回一个 Money 类型的对象，它的值是第一个实参的值减去第二个实参的值。
5. 注意图 11.3 的 Money 类定义中的成员函数 output。为了将 Money 类型的值写到屏幕上，需要在调用 output 时，将 cout 作为它的参数。例如，假定 purse 是 Money 类型的一个对象，为了将 purse 中的金额输出到屏幕，需要在程序中执行以下调用：

   ```
   purse.output(cout);
   ```

   但更好的方案是默认就将输出发送到屏幕，不一定非要提供 cout 这个实参。

   请改写图 11.3 的 Money 类的定义，重载函数名称 output，提供两个版本的 output。一个版本和图 11.3 的相同，另一个版本不取任何参数，直接将输出发送到屏幕。使用新版本 Money 类时，以下两个调用是等价的：

   ```
   purse.output(cout);
   purse.output();
   ```

   但第二个调用更简单。注意，由于 output 函数现在有两个版本，所以你仍能将输出发送到文件。假定 outs 是输出文件流，而且已连接到一个文件，以下调用将 purse 对象中的金额输出到与 outs 连接的文件中：

   ```
   purse.output(outs);
   ```

6. 注意图 11.3 的 Money 类中对成员函数 input 的定义。如用户提供了不正确的输入，函数会报告一条错误消息，并终止程序。例如，假定用户遗漏一个美元符号，函数就会报错。但那里进行的检查不能捕捉所有不正确的输入。例如，负的金额本该以-$9.95 这样的形式输入，但假如用户错误采用$-9.95 的形式，那么虽然输入时不报错，但 Money 对象将被设为不正确的值。如不慎输入$-9.95，成员函数 input 读取的具体是多大的金额？怎样添加附加的检查，捕捉因负号位置不正确导致的错误。
7. 建议写一个小程序来测试前置 0 是否会导致编译器将输入的数字解析成八进制数字。请动手写一个这样的程序。

const 参数修饰符

“传引用”参数效率上优于“传值”参数。传值参数是局部变量，被初始化成实参的值，所以调用函数时会存在实参的两个拷贝。而传引用参数只是占位符，会被实参取代，所以只存在实参的一个拷贝。对于简单类型(比如 int 或 double)的参数，这种效率上的差异可忽略不计。但对于类类型的参数，两者效率上的区别有时就非常明显，必须引起重视。类作为参数使用时，有必要使用传引用参数，而不要使用传值参数，即使函数内部不更改

参数。

如使用传引用参数，而且函数不更改参数，就可为参数做上标记，让编译器知道参数不应更改。为此，要在参数类型前添加修饰符 const。这样的参数是**常量参数**。以图 11.3 定义的 Money 类为例。对于友元函数 add，可像下面这样将 Money 类型的参数设为常量：

```
class Money
{
public:
    friend Money add(const Money& amount1, const Money& amount2);
    // 前条件：amount1 和 amount2 已被赋值
    // 返回 amount1 和 amount2 的值之和
       . . .
```

使用常量参数时，在函数声明和函数定义头中都必须使用修饰符 const，所以像上面那样更改了 add 函数的声明之后，函数定义应该像下面这样开头：

```
Money add(const Money& amount1, const Money& amount2)
{
    ...
```

函数定义剩余部分与图 11.3 相同。

常量参数是自动错误检查的一种形式。函数定义不慎更改了常量参数，编译器会报错。参数修饰符 const 适合任何参数，但通常只将它用于传引用的类参数(偶尔也用于其他一些参数，前提是传递的实参值较大)。

调用函数时，传引用参数被替换成实参，而函数调用也许会(也许不会)更改实参的值。调用成员函数时，调用对象的行为和传引用参数很相似，因为成员函数调用可更改调用对象的值。例如以下语句，其中的 Money 类已在图 11.3 中定义：

```
Money m;
m.input(cin);
```

声明对象 m 之后，成员变量 allCents 的值被初始化为 0。调用成员函数 input 将成员变量 allCents 的值变成基于用户输入的新值。所以，m.input(cin) 调用会更改 m 的值，如同 m 是一个传引用参数。

应用于参数的修饰符 const 以同样的方式应用于调用对象。如果成员函数不应更改调用对象的值，就可用 const 修饰符标记该函数。函数代码不慎更改了调用对象的值，编译器会报错。在成员函数的情况下，关键字 const 要放到函数声明的后面，刚好在末尾的分号之前，如下所示：

```
class Money
{
public:
       . . .
    void output(ostream& outs) const;
       . . .
```

修饰符 const 在函数声明和函数定义中都要使用，所以 output 的函数定义像下面这样开头：

```
void Money::output(ostream& outs) const
{
    ...
```

函数定义剩余的部分与图 11.3 相同。

陷阱：修饰符 const 的用法不一致

视频讲解：*const Confusion*

修饰符 const 要么都用，要么都不用。为特定类型的参数使用了 const 之后，对于同类型的其他所有参数，如果它们不由函数调用更改，那么也应该使用 const。此外，如果该类型是一个类，凡是不更改调用对象值的成员函数都应使用 const 修饰符。之所以要制订这个规矩，原因和函数调用中的函数调用有关。例如以下 guarantee 函数定义：

```
void guarantee(const Money& price)
{
    cout << "If not satisfied, we will pay you\n"
         << "double your money back.\n"
         << "That's a refund of $"
         << (2 * price.getValue()) << endl;
}
```

不为成员函数 getValue 的函数声明加上 const 修饰符，guarantee 函数会在大多数编译器上报错。成员函数 getValue 不更改调用对象 price。但当编译器处理 guarantee 的函数定义时，它认为 getValue 确实(或至少有可能)会更改 price 的值。这是由于当编译器翻译 guarantee 的函数定义时，对于成员函数 getValue，它目前唯一知道的只有 getValue 的函数声明。在函数声明中，如果不包含告诉编译器调用对象不会被更改的 const，编译器就假定调用对象会被更改。所以，一旦为 Money 类型的参数使用了修饰符 const，就要为不更改调用对象值的所有 Money 成员函数都使用 const。尤其是，在成员函数 getValue 的函数声明中，必须包含修饰符 const。

图 11.4 重写了图 11.3 的 Money 类的定义，但这一次，我们在所有合适的地方都使用了 const 修饰符。成员函数和友元函数的定义与图 11.3 相同，只是必须在函数头中使用 const 修饰符，以便与图 11.4 显示的函数声明匹配。■

图 11.4　具有常量参数的 Money 类

```
1   // 该程序用于演示 Money 类
2   class Money
3   {
4   public:
5       friend Money add(const Money& amount1, const Money& amount2);
6       // 前条件：amount1 和 amount2 已被赋值
7       // 返回 amount1 和 amount2 的值之和
8       friend bool equal(const Money& amount1, const Money& amount2);
9       // 前条件：amount1 和 amount2 已被赋值
10      // 如果 amount1 和 amount2 具有相同的值，就返回 true；否则返回 false

11      Money(long dollars, int cents);
12      // 初始化对象，根据参数中的美元和美分数额来表示一个金额
13      // 如果金额是负数，那么美元和美分都必须为负

14      Money(long dollars);
15      // 初始化对象，使它的值表示$dollars.00

16      Money();
17      // 初始化对象，使它的值表示$0.00

18      double getValue() const;
19      // 前条件：调用对象已被赋值
20      // 返回在调用对象的数据中记录的金额
```

```
21        void input(istream& ins);
22        // 前条件：如果 ins 是一个文件输入流，那么 ins 已经连接到一个文件
23        // 在输入流 ins 中，已输入了一个金额，其中包括美元符
24        // 负数金额的表示方式为-$100.00
25        // 后条件：调用对象的值被设为来自输入流 ins 的金额
26        void output(ostream& outs) const;
27        // 前条件：如果 outs 是一个文件输出流，那么 outs 已经连接到一个文件
28        // 后条件：调用对象中记录好的美元符和金额将发送给输出流 outs
29    private:
30        long allCents;
31    };
```

参数修饰符 const

在传引用参数的类型之前添加修饰符 const，该参数就是**常量参数**(函数定义头也应包含一个 const，与函数声明匹配)。添加 const 后，编译器就知道该参数不应更改。在函数定义中错误更改了常量参数，编译器会报错。对于类类型的参数，如果它不会由函数更改，就应该设为传引用参数，而不应设为传值参数。

如成员函数不更改它的调用对象的值，应在函数声明中用修饰符 const 标记该函数。如函数定义有错，造成它更改了调用对象，而且该函数已用 const 标记，编译器就会报错。const 要放到函数声明之后，刚好在末尾的分号之前。函数定义头也应包含一个 const，与函数声明匹配。

示例

```
class Sample
{
public:
    Sample();
    friend int compare(const Sample& s1, const Sample& s2);
    void input();
    void output() const;
  private:
      int stuff;
      double moreStuff;
  };
```

修饰符 const 要么不用，要么都用。任何类参数以及类的成员函数，只要合适，都应使用修饰符 const。如果 const 不是每次都适合一个类，就永远不要为那个类使用。

√ 自测题

8. 给出成员函数 getValue 的完整定义，与图 11.4 的 Money 定义配合使用。

9. 在图 11.4 的 Money 类的成员函数 input 的函数声明中，像下面这样添加修饰符 const 为什么不正确？

```
class Money
{
        . . .
public:
    void input(istream& ins) const;
        . . .
}
```

10. 传值参数和传 const 引用的参数有什么相同点和不同点？下面的函数声明对这两种调用进行了演示：

```
void callByValue(int x);
void callByConstReference(const int& x);
```

11. 给定以下定义：

```
const int x = 17;
```

```
class A
{
public:
    A();
    A(int x);
    int f() const;
    int g(const A& x);
private:
    int i;
};
```

对于其中三个 const 关键字，每个都相当于对编译器的一个承诺，而且编译器会强制这个承诺。那么，每种情况下的承诺具体是什么？

11.2　重载操作符

> 用一个名称来替代不同的事物，这门艺术叫数学。用不同的名称来意指同一个事物，那是诗人干的事。
>
> ——*儒勒·亨利·庞加莱*[①]

本章前面展示了如何使 add 函数成为 Money 类的友元，并用它将 Money 类型的两个对象加到一起(图 11.3)。add 函数确实能完成对象相加的任务，但更好的方式是使用普通的操作符+对 Money 类型的两个值进行相加，如以下代码的最后一行所示：

```
Money total, cost, tax;
cout << "Enter cost and tax: ";
cost.input(cin);
tax.input(cin);
total = cost + tax;
```

而不是使用下面这种比较繁琐的写法：

```
total = add(cost, tax);
```

记住，像+这样的操作符本质上是函数，只是语法稍微有别于普通函数。在普通函数调用中，实参要放在函数名之后的一对圆括号中，如下所示：

```
add(cost, tax)
```

但对于(二元)操作符，实参要放在操作符两侧，示例如下：

```
cost + tax
```

函数能重载以获取不同类型的实参。操作符是函数，所以操作符也能重载。重载操作符(比如+)的方式与重载函数名的方式大致相同。本节介绍如何在 C++中重载操作符。

重载操作符

可重载操作符+(以及其他许多操作符)来接受类类型的实参。重载操作符+与定义函数 add(在图 11.3 中给出)的区别只涉及语法上的一处微小变动。对于重载的操作符+，它的定义与函数 add 的定义基本相同。唯一区别是要使用名称+，而不是名称 add。另外，要在操作符+之前附加关键字 operator。图 11.5 重写了 Money 类型，在其中包括重载的操作符+，

---

① Jules Henri Poincaré(1854—1912)，法国最伟大的数学家，理论科学和科学哲学家，公认的 19 世纪后期和 20 世纪初期的领袖数学家。他对数学、数学物理和天体力学做出了很多创造性的基础贡献。——译注

并将定义嵌入一个小的演示程序中。

图 11.5 的 Money 类还重载了操作符==，以便用它比较 Money 类型的两个对象。假定 amount1 和 amount2 是 Money 类型的两个对象，我们希望以下表达式：

```
amount1 == amount2
```

返回与以下布尔表达式相同的值：

```
amount1.allCents == amount2.allCents
```

如图 11.5 所示，我们在重载操作符==时，返回的正是这个值。

图 11.5　重载操作符

```
1  // 该程序用于演示 Money 类(这是基于图 11.3 原始版本和图 11.4 重写版本的改进版本)
2  #include <iostream>
3  #include <cstdlib>
4  #include <cctype>
5  using namespace std;
6
7  // 用于表示美元金额的类
8  class Money
9  {
10 public:
11     friend Money operator +(const Money& amount1, const Money& amount2);
12     // 前条件：amount1 和 amount2 已被赋值
13     // 返回 amount1 和 amount2 的值之和

14     friend bool operator ==(const Money& amount1, const Money& amount2);
15     // 前条件：amount1 和 amount2 已被赋值
16     // 如果 amount1 和 amount2 具有相同的值，就返回 true；否则返回 false

17     Money(long dollars, int cents);

18     Money(long dollars);

19     Money();

20     double getValue() const;

21     void input(istream& ins);

22     void output(ostream& outs) const;
23 private:
24     long allCents;
25 };

   <来自图 11.3 的任何额外函数声明都放在这里>

26 int main()
27 {
28     Money cost(1, 50), tax(0, 15), total;
29     total = cost + tax;

30     cout << "cost = ";
31     cost.output(cout);
32     cout << endl;
33     cout << "tax = ";
34     tax.output(cout);
35     cout << endl;
36     cout << "total bill = ";
37     total.output(cout);
38     cout << endl;
39     if (cost == tax)
40         cout << "Move to another state.\n";
41     else
42         cout << "Things seem normal.\n";
43     return 0;
44 }
```

为节省篇幅，这里省略了图 11.4 中的部分注释。但在实际的应用程序中，应该包括它们

```
45
46 Money operator +(const Money& amount1, const Money& amount2)
47 {
48     Money temp;
49     temp.allCents = amount1.allCents + amount2.allCents;
50     return temp;
51 }
52
53 bool operator ==(const Money& amount1, const Money& amount2)
54 {
55     return (amount1.allCents == amount2.allCents);
56 }
```

<成员函数定义与图 11.3 相同，只是需要在函数头中添加 const，使函数头与上述类定义中的函数声明相匹配。成员函数定义不需要进行其他更改。成员函数主体与图 11.3 完全一致>

示范输出

```
cost = $1.50
tax = $0.15
total bill = $1.65
Things seem normal.
```

操作符重载

包括+，-，/，%等在内的(二元)操作符本质上是函数。调用这种“函数”时，要以不同的语法列出实参。操作符函数的实参要在操作符前后列出；普通函数的实参则在函数名之后的圆括号中列出。操作符函数的定义与普通函数相似，只是要在操作符之前附加保留字 operator。可为类类型重载预定义操作符(比如+)，为这些操作符赋予新含义。

虽然并非必须，但操作符可作为类的友元。图 11.5 展示了如何将操作符+重载为友元。

大多数(但并非全部)操作符都可重载。操作符不一定是类的友元，但一般情况下都希望如此。参见补充内容“操作符重载规则”，了解与操作符重载的时机和方式。

操作符重载规则

- 重载操作符时，至少一个实参必须是类类型。
- 重载的操作符可以是(但不一定是)类的友元；操作符函数可以是类的成员，也可以是普通(非友元)函数(附录 8 讨论如何将操作符重载为类的成员)。
- 不能新建操作符。只能重载现有操作符，比如+，-，*，/，%等。
- 不能改变操作符获取的实参数量。例如，重载%时，不能把它从二元操作符变成一元操作符；重载++时，不能把它从一元操作符变成二元操作符。
- 不能改变操作符的优先级。重载的操作符具有和原始版本一样的优先级。例如，x * y + z 总是表示 (x * y) + z，即使 x，y 和 z 是对象，而且操作符+和*已针对相应的类进行了重载。
- 以下操作符不可重载：圆点操作符(.)、作用域解析操作符(::)以及本书没有讨论的操作符 .*和?:。
- 虽然赋值操作符=能重载，将它的默认含义转变成其他含义，但方式不同于本节描述的方式。本章最后的“重载赋值操作符”一节将详细讨论如何重载操作符=。另一些操作符，包括 [] 和->，也必须采取不同于本章描述的方式进行重载。操作符 [] 和->将在本书以后讨论。

√ 自测题

12. (二元)操作符和函数有什么区别？
13. 假定要重载操作符<，使其应用于图 11.5 的 Money 类型。要在图 11.5 的 Money 定义中添加什么内容？
14. 假定要重载操作符<=，使其应用于图 11.5 的 Money 类型。要在图 11.5 的 Money 定义中添加什么内容？
15. 可通过操作符重载来改变+之于整数的行为吗？请说明原因。

用于自动类型转换的构造函数

如果类定义包含了恰当的构造函数，系统会自动执行特定的类型转换。例如，如果程序包含图 11.5 给出的 Money 类定义，可以在程序中使用下面这些语句：

```
Money baseAmount(100, 60), fullAmount;
fullAmount = baseAmount + 25;
fullAmount.output(cout);
```

输出结果如下：

```
125.60
```

代码看起来简单和自然，但存在一个容易被忽视的问题。在表达式 baseAmount + 25 中，25 的类型不合适。图 11.5 只重载了操作符+，使其可用于 Money 类型的两个值。但没有重载操作符+，使其获取一个 Money 类型的值和一个整数作为参数。常量 25 是整数，不是 Money 类型。常量 25 可以是 int 或 long，但不能作为 Money 值使用，除非类定义告诉系统如何将整数转换成 Money 值。为了让系统知道 25 的意思是$25.00，唯一的办法就是包括一个构造函数，它获取 long 类型的一个参数。一旦系统看到以下表达式：

```
baseAmount + 25
```

首先就会检查操作符+是否已针对 Money 类型的值和整数值的组合进行了重载。由于没有进行这种重载，所以系统接着检查是否有一个构造函数获取单个整数值参数。发现这样的构造函数，就用那个构造函数将整数 25 转换成 Money 类型的值。获取单个 long 值作为参数的构造函数告诉系统如何将整数(比如 25)转换成 Money 值。单参数构造函数指出 25 应该转换成 Money 类型的一个对象，该对象的成员变量 allCents 等于 2500；换言之，构造函数将 25 转换成 Money 类型的对象，该对象代表$25.00(构造函数的定义参见图 11.3)。

注意，除非有合适的构造函数，否则类型转换不会成功。例如，图 11.5 的 Money 类没有能获取单个 double 值的构造函数，所以假如将 baseAmount 和 fullAmount 声明为 Money 类型的对象，以下语句是非法的，将产生一条错误消息：

```
fullAmount = baseAmount + 25.67;
```

为了使上述语句合法，可更改 Money 类的定义添加另一个构造函数。需添加的构造函数的声明如下所示：

```
class Money
{
public:
        . . .
    Money(double amount);
```

```
// 初始化对象，让它的值表示$amount
    . . .
```

再为这个新的构造函数写一个定义，详情参见自测题 16。

表面上这些自动类型转换(由构造函数负责)最适合数值操作符(比如+和-) 的重载。但事实上，可采取同样的方式，将这些自动转换应用于普通函数的实参、成员函数的实参以及其他重载操作符的实参。

√ 自测题

16. 针对本节最后讨论的构造函数，请给出它的定义。构造函数要添加到图 11.5 的 Money 类定义中。定义像下面这样开头：

```
Money::Money(double amount)
{
```

重载一元操作符

除了二元操作符(比如 x + y 中的+)，还有一元操作符(比如求反操作符-)。在下面的语句中，一元操作符-将变量 x 的值设置成变量 y 的值的相反数：

```
x = -y;
```

递增操作符++和递减操作符--也是一元操作符。

可以像重载二元操作符那样重载一元操作符。例如，可重新定义图 11.5 的 Money 类型，为操作符-同时提供一元和二元操作符版本。图 11.6 展示了类的最新定义。假定程序包含这个类定义和以下代码：

```
Money amount1(10), amount2(6), amount3;
```

则以下语句将 amount3 的值设为 amount1 减去 amount2 的值：

```
amount3 = amount1 - amount2;
```

随后，以下语句在屏幕上输出$4.00：

```
amount3.output(cout);
```

另一方面，以下语句将 amount3 的值设置成 amount1 的相反数：

```
amount3 = -amount1;
```

随后，以下语句在屏幕上输出-$10.00：

```
amount3.output(cout);
```

可采取类似的方式重载操作符++和--。重载的定义适用于操作符的前缀版本，比如++x 和--x。++和--的后缀版本(比如 x++和 x--)要用不同的方式处理，但我们不准备讨论它们(嘿，入门课程不可能面面俱到吧)。

图 11.6　重载一元操作符

```
1  // 演示如何在 Money 类中重载一元操作符-        这是图 11.5 中 Money 类的改进版本
2  class Money
3  {
4  public:
```

```
5       friend Money operator +(const Money& amount1, const Money& amount2);
6       friend Money operator -(const Money& amount1, const Money& amount2);
7       // 前条件：amount1 和 amount2 已被赋值
8       // 返回 amount1 减去 amount2 的结果
9       friend Money operator -(const Money& amount);
10      // 前条件：amount 已被赋值
11      // 返回 amount 值的反数
12      friend bool operator ==(const Money& amount1, const Money& amount2);
13      Money(long dollars, int cents);
14      Money(long dollars);
15      Money();
16      double getValue() const;
17      void input(istream& ins);
18      void output(ostream& outs) const;
19  private:
20      long allCents;
21  };
    <在此添加其他函数声明和程序的 main 部分>
22  Money operator -(const Money& amount1, const Money& amount2)
23  {
24      Money temp;
25      temp.allCents = amount1.allCents - amount2.allCents;
26      return temp;
27  }
28  Money operator -(const Money& amount)
29  {
30      Money temp;
31      temp.allCents = -amount.allCents;
32      return temp;
33  }
    <其他函数定义与图 11.5 中相同>
```

为节省篇幅，这里省略了 include 预编译指令和部分注释，但在实际的程序中，应将它们包含在内

重载>>和<<

为 cout 使用的插入操作符<<是二元操作符。例如以下语句：

```
cout << "Hello out there.\n";
```

操作符是<<，第一个操作数是输出流 cout，第二个则是字符串值"Hello out there.\n"。这两个操作数都可以改变。如 fout 是 ofstream 类型的输出流，而且已通过 open 调用与一个文件连接，就可将 cout 替换成 fout，字符串会写到与 fout 连接的文件中。当然，还可以将字符串"Hello out there.\n"替换成其他字符串、变量或数字。由于<<是操作符，所以应该能像重载+和-等操作符那样重载<<。这个推论是正确的，但在重载输入和输出操作符 (>>和<<) 时，要注意更多的细节。

Money 类之前的定义一直在用成员函数 output 输出 Money 类型的值。这虽然可行，但用插入操作符<<输出更好，如以下语句所示：

```
Money amount(100);
cout << "I have " << amount << " in my purse.\n";
```

这样一来，就用不着像下面这样使用成员函数 output：

```
Money amount(100);
cout << "I have ";
amount.output(cout);
cout << " in my purse.\n";
```

重载操作符<<时，必须决定在下面这样的一个表达式中，<<应该返回什么：

```
cout << amount
```

表达式的两个操作数是 cout 和 amount。对表达式进行求值，应导致 amount 的值写到屏幕。但是，如果<<是+或*那样的操作符，上述表达式还应返回一个值。例如，n1 + n2 这个表达式就会返回一个值。但是，cout << amount 返回什么呢？为了回答这个问题，需要分析使用了<<的一个更复杂的表达式。

下面对一个使用<<的表达式链进行求值：

```
cout << "I have " << amount << " in my purse.\n";
```

将操作符<<视为与其他操作符(比如+)相似的操作符，上述表达式应该(而且确实)等价于下面这个表达式：

```
((cout << "I have ") << amount) << " in my purse.\n";
```

操作符<<返回什么值，上述表达式才显得有意义呢？第一个求值的是下面这个子表达式：

```
(cout << "I have ")
```

要正常工作，上述子表达式最好返回 cout，使整个表达式变成下面这样：

```
(cout << amount) << " in my purse.\n";
```

同样地，要继续正常工作，(cout << amount)也最好返回 cout，使整个表达式变成下面这样：

```
cout << " in my purse.\n";
```

图 11.7 对此进行了演示。操作符<<应返回它的第一个实参，也就是 ostream 类型的一个流。

图 11.7　<<作为操作符

```
1   cout << "I have " << amount << " in my purse.\n";
2
3   等价于：
4
5   ((cout << "I have ") << amount) << " in my purse.\n";
6
7   它像下面这样求值：
8
9   首先求值的是(cout << "I have ")，它返回 cout：
10  ((cout << "I have ") << amount) << " in my purse.\n";
11
12                 输出字符串"I have "
13
14  (cout << amount) << " in my purse.\n";
15
16
17  然后求值(cout << amount)，它返回 cout：
18
19  (cout << amount) << " in my purse.\n";
20
21              输出 amount 的值
22
23  cout << " in my purse.\n";
24
25
```

```
26  再求值 cout << " in my purse.\n"，它返回 cout：
27
28  cout << " in my purse.\n";
29
30
31                        输出字符串
32  cout;                 "in my purse.\n"

                没有更多<<了，所以过程终止
```

所以，为了和 Money 类配合使用，重载操作符<<应该像下面这样声明：

```
class Money
{
public:
     …
    friend ostream& operator <<(ostream& outs, const Money& amount);
    // 前条件：如果 outs 是一个文件输出流，那么 outs 已经连接到一个文件
    // 后条件：调用对象中记录的美元符号和金额将发送给输出流 outs
      ⋮
```

重载了插入(输出)操作符<<之后，就不再需要成员函数 output，可将 output 从 Money 类的定义中删除。重载的操作符<<的定义与成员函数 output 非常相似，大致如下：

```
ostream& operator <<(ostream& outs, const Money& amount)
{
    <这部分与图 11.3 的 Money::output 的主体相同，
    不过要将 allCents 替换成 amount.allCents>

    return outs;
}
```

上述声明和定义有一点需要注意。返回类型 ostream&中的&是什么意思？简单地回答，只要操作符(或函数)返回流，就必须在返回类型名称末尾添加&。这个简单的规则让你能够重载操作符<<和>>。然而，虽然这是编写类定义和程序时的一个行之有效的规则，但不是特别令人满意。虽然不硬性要求了解&的真正含义，但如果稍微解释一下它，就会明白这个规则为什么要求添加一个&。

为返回类型的名称添加&，意思是操作符(或函数)需要返回引用。前面讲的所有函数和操作符都只是返回值。然而，如果返回类型是流，就不能简单地返回流的值。对于一个流，它的值可能是一个完整文件、键盘或屏幕，返回那些东西可能是没有意义的。所以，我们希望只返回流本身，而不是返回流的值。为返回类型的名称添加&，就表明操作符(或函数)要返回**引用**。这表示返回的是对象本身，而非对象的值。

提取操作符>>的重载方式类似于插入操作符<<。但是，对于提取(输入)操作符，第二个参数是接收输入值的对象，所以第二个参数必须是标准的传引用参数。重载的提取操作符>>大致像下面这样定义：

```
istream& operator >>(istream& ins, Money& amount)
{
     <这部分与图 11.3 的 Money::input 主体相同，
     不过要将 allCents 替换成 amount.allCents>

    return ins;

}
```

重载>>和<<

输入和输出操作符>>和<<可以像其他操作符那样重载。返回值必须是流。返回值类型名称后必须添加符号&。函数声明和函数定义头如下所示。图 11.8 展示了一个例子。

函数声明

operator 是重载操作符时必须使用的保留字

这个参数指定了要接收输入的对象

```
class ClassName
{
public:
      . . .
    friend istream& operator >>(istream& Parameter_1,
                                 ClassName& Parameter_2);
    friend ostream& operator <<(ostream& Parameter_3,
                                 const ClassName& Parameter_4);
      . . .
```

定义

```
istream& operator >>(istream& Parameter_1,
                     ClassName& Parameter_2);
{
      . . .
}
ostream& operator <<(ostream& Parameter_3,
                     const ClassName& Parameter_4);
{
      . . .
```

图 11.8 给出了重载的操作符<<和>>的完整定义，我们再次重写了 Money 类。这一次主要对操作符<<和>>进行了重载，允许为 Money 类型的值使用这些操作符。

图 11.8　重载<<和>>

这是图 11.6 的 Money 类的改进版本

尽管这里省略了图 11.5 和图 11.6 的部分注释，但在实际程序中，应将它们包含在内

```
1   // 该程序用于演示 Money 类
2   #include <iostream>
3   #include <fstream>
4   #include <cstdlib>
5   #include <cctype>
6   using namespace std;
7
8   // 表示美元金额的类
9   class Money
10  {
11  public:
12      friend Money operator +(const Money& amount1, const Money& amount2);
13      friend Money operator -(const Money& amount1, const Money& amount2);
14      friend Money operator -(const Money& amount);
15      friend bool operator ==(const Money& amount1, const Money& amount2);
16      Money(long dollars, int cents);
17      Money(long dollars);
18      Money();
19      double getValue() const;
20      friend istream& operator >>(istream& ins, Money& amount);
21      // 重载>>操作符，使其能支持 Money 类型的输入值
22      // 负数金额的输入格式是-$100.00
23      // 前条件：如果 ins 是一个文件输入流，那么 ins 已经连接到一个文件
25      friend ostream& operator <<(ostream& outs, const Money& amount);
26      // 重载<<操作符，使其能支持 Money 类型的输出值
27      // 为 Money 类型的每个输出值都附加一个美元符号前缀
28      // 前条件：如果 outs 是一个文件输出流，那么 outs 已经连接到一个文件
```

```
29  private:
30      long allCents;
31  };
32  int digitToInt(char c);
33  // 在重载的输入操作符>>的定义中使用
34  // 前条件：c是'0'～'9'的一个数字，返回该数字的整数值；例如，digitToInt('3')返回3
35
36  int main()
37  {
38      Money amount;
39      ifstream inStream;
40      ofstream outStream;
41
42      inStream.open("infile.dat");
43      if (inStream.fail())
44      {
45          cout << "Input file opening failed.\n";
46          exit(1);
47      }
48
49      outStream.open("outfile.dat");
50      if (outStream.fail())
51      {
52          cout << "Output file opening failed.\n";
53          exit(1);
54      }
55
56      inStream >> amount;
57      outStream << amount
58                << " copied from the file infile.dat.\n";
59      cout << amount
60           << " copied from the file infile.dat.\n";
61
62      inStream.close();
63      outStream.close();
64
65      return 0;
66  }
67  // 使用iostream，cctype，cstdlib：
68  istream& operator >>(istream& ins, Money& amount)
69  {
70      char oneChar, decimalPoint,
71           digit1, digit2; // 美元金额的数字
72      long dollars;
73      int cents;
74      bool negative;// 如果输入为负数，则设为true
75      ins >> oneChar;
76      if (oneChar == '-')
77      {
78          negative = true;
79          ins >> oneChar; // 读入'$'
80      }
81      else
82          negative = false;
83      // 如果输入是合法值，那么oneChar == '$'
84      ins >> dollars >> decimalPoint >> digit1 >> digit2;
85      if ( oneChar != '$' || decimalPoint != '.'
86          || !isdigit(digit1) || !isdigit(digit2) )
87      {
88          cout << "Error illegal form for money input\n";
89          exit(1);
90      }
91      cents = digitToInt(digit1)*10 + digitToInt(digit2);
92      amount.allCents = dollars*100 + cents;
93      if (negative)
94          amount.allCents = -amount.allCents;
95      return ins;
```

```
96  }
97
98  int digitToInt(char c)
99  {
100     return ( int(c) - int('0') );
101 }
102 // 使用 cstdlib 和 iostream:
103 ostream& operator <<(ostream& outs, const Money& amount)
104 {
105     long positiveCents, dollars, cents;
106     positiveCents = labs(amount.allCents);
107     dollars = positiveCents/100;
108     cents = positiveCents % 100;
109
110     if (amount.allCents < 0)
111        outs << "-$" << dollars << '.';
112     else
113        outs << "$" << dollars << '.';
114
115     if (cents < 10)
116        outs << '0';
117     outs << cents;
118
119     return outs;
120 }
121
    <成员函数和其他重载的操作符的定义放到这里。详细定义参见图 11.3、图 11.4、图 11.5 和图 11.6>
```

infile.dat (不会被程序更改)	outfile.dat (程序运行之后)
$1.11　$2.22 $3.33	$1.11 copied from the file infile.dat.

屏幕输出

```
$1.11 copied from the file infile.dat.
```

√ 自测题

17. 下面是名为 Pairs 类的定义。Pairs 类型的对象可在需要有序数偶(ordered pair)的任何情况下使用。你的任务是写重载的操作符>>和<<的定义，使 Pairs 类的对象能支持 (5,6) (5,-4) (-5,4) 或者 (-5,-6) 等输入/输出形式。不需要实现任何构造函数或者其他成员，也不需要检查输入格式。

```
#include <iostream>
using namespace std;
class Pairs
{
public:
    Pairs();
    Pairs(int first, int second);
    // 其他成员和友元
    friend istream& operator >> (istream& ins, Pairs& second);
    friend ostream& operator << (ostream& outs, const Pairs& second);
private:
    int f;
    int s;
};
```

18. 下面是名为 Percent 类的定义。Percent 类型的对象代表 10%或 99%这样的百分比。请给出重载的操作符>>和<<的定义，使它们能在 Percent 类的对象的输入和输出中使用。假定输入总是由一个整数加一个字符'%'构成，比如 25%。假定所有百分数都是整数(也就是说，不会出现 25.5%这样的百分比)，并存储在 int 成员变量 value 中。不需要定义其他重载的操作符，也不需要定义构造函数。只需定义重载的操作符>>和<<。

```
#include <iostream>
using namespace std;

class Percent
{
public:
     friend bool operator ==(const Percent& first, const Percent& second);

    friend bool operator <(const Percent& first, const Percent& second);

    Percent();

    Percent(int percentValue);

    friend istream& operator >>(istream& ins, Percent& theObject);
    // 重载操作符>>，使其支持 Percent 类型的输入值
    // 前条件：如果 ins 是一个文件输入流，那么 ins 已经连接到一个文件

    friend ostream& operator <<(ostream& outs, const Percent& aPercent);
    // 重载操作符<<，使其支持 Percent 类型的输出值
    // 前条件：如果 outs 是一个文件输出流，那么 outs 已经连接到一个文件
private:
    int value;
};
```

11.3　数组和类

数组、结构和类可组合成复杂的结构化类型，比如结构数组、类数组以及数组作为成员变量的类。本节用一些简单的例子演示这些可能性。

类数组

数组基类型可为任意类型，包括自定义类型，比如结构和类类型。如希望每个索引变量(即数组元素)都包含不同类型的数据项，可以使用结构数组。例如，假定要用数组容纳 10 个天气“数据点”(data point)，每个数据点都由风速和风向(北、南、东或西)构成，就可以使用以下类型定义和数组声明：

```
struct WindInfo
{
    double velocity; // 代表风速，单位是英里/小时(MPH)
    char direction; // 代表风向，可为'N'，'S'，'E'或者'W'
};

WindInfo dataPoint[10];
```

使用以下 for 循环填充 dataPoint 数组：

```
int i;
for (i = 0; i < 10; i++)
{
    cout << "Enter velocity for "
         << i << " numbered data point: ";
    cin >> dataPoint[i].velocity;
    cout << "Enter direction for that data point"
         << " (N, S, E, or W): ";
    cin >> dataPoint[i].direction;
}
```

读 dataPoint[i].velocity 这样的表达式时，必须从左向右读，而且必须非常小心。首先，dataPoint 是数组。所以，dataPoint[i]是数组中索引为 i 的索引变量。数组的所

有索引变量都具有 WindInfo 类型。而 WindInfo 是结构，其中含有两个成员变量，分别为 velocity(风速)和 direction(风向)。因此，dataPoint[i].velocity 必须理解成索引为 i 的数组元素的成员变量 velocity。通俗地说，dataPoint[i].velocity 是编号为 i 的数据点的风速。类似地，dataPoint[i].direction 是编号为 i 的数据点的风向。

dataPoint 数组的 10 个数据点可用以下 for 循环写到屏幕：

```
for (i = 0; i < 10; i++)
    cout << "Wind data point number " << i << ": \n"
         << dataPoint[i].velocity
         << " miles per hour\n"
         << "direction " << dataPoint[i].direction
         << endl;
```

图 11.9 展示了 Money 类的定义。Money 类的对象代表美元金额。类的成员函数、成员操作和友元函数的定义可在图 11.3～图 11.8 以及自测题 13 的答案中找到。可以使用基类型为 Money 的数组。图 11.9 给出了一个简单例子。程序读入 5 个金额，计算每个金额与最大金额的差值。注意，处理基类型为类的数组和处理其他数组没有多大分别。事实上，图 11.9 的程序和图 7.1 的程序非常相似，只是在图 11.9 中基类型是类。

声明类数组时，会调用默认构造函数初始化索引变量。所以，任何要作为数组基类型使用的类，都应该有默认构造函数。类数组的使用方式与 int 或 double 数组一样。例如，每个金额与最大金额的差值存储在名为 difference 的数组中，如下所示：

```
Money difference[5];
for( i = 0; i < 5; i++)
    difference[i] = max - amount[i];
```

图 11.9　使用了 Money 对象数组

```
1   // 这是 Money 类的定义
2   // 这个类型的值表示美元金额
3   #include <iostream>
4   using namespace std:

5   class Money
6   {
7   public:
8       friend Money operator +(const Money& amount1, const Money& amount2);
9       // 返回 amount1 和 amount2 的值之和

10      friend Money operator -(const Money& amount1, const Money& amount2);
11      // 返回 amount1 减去 amount2 的结果

12       friend Money operator -(const Money& amount);
13      // 返回 amount 值的反数

14       friend bool operator ==(const Money& amount1, const Money& amount2);
15       // 如果 amount1 和 amount2 具有相同的值，就返回 true，否则返回 false

16      friend bool operator < (const Money& amount1, const Money& amount2);
17      // 如果 amount1 小于 amount2，就返回 true，否则返回 false

18      Money(long dollars, int cents);
19      // 初始化对象，使它的值代表一个金额；该金额的美元数和美分数由参数来提供
20       // 如果金额为负，dollars 和 cents 都应该为负

21      Money(long dollars);
22       // 初始化对象，使它的值代表$dollars.00

23      Money();
```

```
24      // 初始化对象，使它的值代表$0.00
25      double getValue() const;
26      // 返回在调用对象的数据部分中记录的金额
27      friend istream& operator >>(istream& ins, Money& amount);
28      // 重载操作符>>，使其能支持 Money 类型的输入值
29      // 负数金额的输入格式是-$100.00
30      // 前条件：如果 ins 是一个文件输入流，那么 ins 已经连接到一个文件
31
32      friend ostream& operator <<(ostream& outs, const Money& amount);
33      // 重载操作符<<，使其能支持 Money 类型的输出值
34      // 为 Money 类型的每个输出值都附加一个美元符号前缀
35      // 前条件：如果 outs 是一个文件输出流，那么 outs 已经连接到一个文件
36    private:
37      long allCents;
38    };
39
    <成员函数和重载的操作符的定义在此>
40  // 读入 5 个金额，显示每个金额与最大金额的差值
41  int main()
42  {
43      Money amount[5], max;
44      int i;
45      cout << "Enter 5 amounts of money:\n";
46      cin >> amount[0];
47      max = amount[0];
48      for (i = 1; i < 5; i++)
49      {
50          cin >> amount[i];
51          if (max < amount[i])
52              max = amount[i];
53          // max 是 amount[0], ..., amount[i]中最大的
54      }
55      Money difference[5];
56      for (i = 0; i < 5; i++)
57          difference[i] = max - amount[i];
58      cout << "The highest amount is " << max << endl;
59      cout << "The amounts and their\n"
60           << "differences from the largest are:\n";
61      for (i = 0; i < 5; i++)
62      {
63          cout << amount[i] << " off by "
64               << difference[i] << endl;
65      }
66      return 0;
67  }
```

示范对话

```
Enter 5 amounts of money:
$5.00 $10.00 $19.99 $20.00 $12.79
The highest amount is $20.00
The amounts and their
differences from the largest are:
$5.00 off by $15.00
$10.00 off by $10.00
$19.99 off by $0.01
$20.00 off by $0.00
$12.79 off by $7.21
```

√ 自测题

19. 为名为 Score 的结构给出类型定义，结构有两个 int 成员变量：homeTeam 和 opponent。声明名为 game 的数组，其中含有 Score 类型的 10 个元素。game 数组记录球队全部 10 场比赛每一场的比分。

20. 写程序读取 5 个金额，使每个金额都增加一倍，然后将倍增的值写到屏幕。使用基类型为 Money 的数组。**提示：**参考图 11.9，但这个程序应该比图 11.9 简单。

数组作为类成员

结构或类可将数组作为自己的成员变量使用。例如，假定你是游泳运动员，想用程序跟踪自己游不同距离时的练习成绩。可用 myBest 结构(具有如下所示的 Data 类型)记录一个距离(单位：m)以及 10 次游过那个距离的每一次的时间(单位：s)。

```
struct Data
{
    double time[10];
    int distance;
};

Data myBest;
```

声明的 myBest 结构有两个成员变量。一个是 int 变量 distance，用于记录距离。另一个是 time 数组，包含 10 个 double 类型的值(用于容纳 10 次游过给定距离的每一次的时间)。为了将 distance 设置成 20 (m)，可使用以下语句：

```
myBest.distance = 20;
```

为了使用从键盘输入的值填充 10 个数组元素，可使用以下 for 循环：

```
cout << "Enter ten times (in seconds):\n";
for (int i = 0; i < 10; i++)
     cin >> myBest.time[i];
```

表达式 myBest.time[i]从左向右读：myBest 是结构，myBest.time 是名为 time 的成员变量。由于 myBest.time 是数组，所以需添加索引。因此，表达式 myBest.time[i]代表数组 myBest.time 的索引为 i 的索引变量。如使用类而不是结构类型，就可使用成员函数来执行所有数组操作，避免这种容易让人混淆的表达式。下一节将对此进行演示。

编程实例　用于部分填充数组的类

图 11.10 显示了 TemperatureList 类的定义，它的对象是温度列表。可在天气分析程序中使用 TemperatureList 类型的对象。温度列表保存在成员变量 list 中，list 是数组。由于这个数组通常只需部分填充，所以要用另一个成员变量 size 跟踪数组中实际使用了多大一部分。size 的值就是 list 数组中实际存储了值的索引变量的数目。

声明 TemperatureList 类型的对象，与声明其他任何类型的对象没有区别。例如，以下语句将 myData 声明为 TemperatureList 类型的对象：

```
TemperatureList myData;
```

这个声明为新对象 myData 调用默认构造函数，对 myData 对象进行初始化，使成员变量

size 具有值 0，表明是空列表。

声明了 myData 这样的对象后，接着可在温度列表中添加数据项(添加到成员数组 list 中)。为此，要像下面这样调用成员函数 addTemperature：

```
myData.addTemperature(77);
```

事实上，这是在 myData 列表中添加温度的唯一途径，因为 list 数组是私有成员变量。注意，通过调用成员函数 addTemperature 添加数据项时，函数调用首先测试数组列表是否已满。数组未满才添加新值。

图 11.10　包含数组成员的类

```
1   // TemperatureList 类的定义
2   // 这个类型的值是华氏温度的列表
3
4   #include <iostream>
5   #include <cstdlib>
6   using namespace std;
7
8   const int MAX_LIST_SIZE = 50;
9
10  class TemperatureList
11  {
12   public:
13      TemperatureList();
14      // 将对象初始化成空列表
15
16      void addTemperature(double temperature);
17      // 前条件：列表未满
18      // 后条件：将温度添加到列表
19
20      bool full() const;
21      // 如果列表已经满了，就返回 true，否则返回 false
22
23      friend ostream& operator <<(ostream& outs,
24         const TemperatureList& theObject);
25      // 重载<<操作符，使其支持 TemperatureList 类型的输出值。每行输出一个温度
26      // 前条件：如果 outs 是一个文件输出流，那么 outs 已经连接到一个文件

27   private:
28      double list[MAX_LIST_SIZE]; // list 数组存储的是华氏温度
29      int size; // 实际填充的数组位置数目
30   };
31
32   // 这是 TemperatureList 类的实现
33   TemperatureList::TemperatureList() : size(0)
34   {
35       // 主体有意留空
36   }
37   void TemperatureList::addTemperature(double temperature)
38   {// 使用 iostream 和 cstdlib:
39       if ( full() )
40       {
41           cout << "Error: adding to a full list.\n";
42           exit(1);
43       }
44       else
45       {
45           list[size] = temperature;
46           size = size + 1;
47       }
48   }

49   bool TemperatureList::full() const
50   {
```

```
51        return (size == MAX_LIST_SIZE);
52    }

53    // 使用 iostream:
54    ostream& operator <<(ostream& outs, const TemperatureList& theObject)
55    {
56        for (int i = 0; i < theObject.size; i++)
57            outs << theObject.list[i] << " F\n";
58        return outs;
59    }
```

TemperatureList 类非常具体。对于 TemperatureList 类的对象，唯一能做的就是将列表初始化成空列表，在列表中添加数据项，检查列表是否已满，以及输出列表。要输出 myData 对象(已在前面声明)中存储的温度，可以像下面这样调用：

```
cout << myData;
```

对于 TemperatureList 类，不可从温度列表(数组)中删除温度。但可删除整个列表，并调用默认构造函数，重新从空列表开始，如下所示：

```
myData = TemperatureList();
```

TemperatureList 类实际并不涉及温度特有的东西。完全可以定义相似的类来表示气压、距离或其他 double 值列表。为了避免分别定义所有这些类的麻烦，可考虑设计一个不那么具体的类，它能表示任意 double 值列表，不指定值具体代表什么。

√ 自测题

21. 修改图 11.10 给出的 TemperatureList 类，添加名为 getSize 的成员函数，它没有任何参数，返回列表中的温度值的数目。

22. 修改图 11.10 给出的 TemperatureList 类，添加名为 getTemperature 的成员函数，它获取一个 int 参数。该参数是大于或等于 0，但小于 MAX_LIST_SIZE 的整数，代表一个列表位置。函数返回 double 类型的值，它是列表中那个位置的温度。例如，如果参数值是 0，那么 getTemperature 返回第一个温度；参数值是 1，返回第二个温度，依此类推。假定 getTemperature 的参数所指定的位置确实存储了一个温度。

11.4　类和动态数组

最温暖舒适的环境之中，……

——*威廉·莎士比亚，《亨利四世》第 3 幕第 1 场*

动态数组的基类型可以是类，类的成员变量也可以是动态数组。可通过任何方式综合运用前面所学的类和动态数组技术。使用类和动态数组时，虽然要留意一些新问题，但基本技术前面都已讲过。下面先分析一个例子。

编程实例　字符串变量类

第 8 章介绍了如何定义用于容纳 C 字符串的数组变量。通过前一小节的学习，你知道了怎样定义动态数组，以便在程序运行时确定数组的长度。本例将定义名为 StringVar 的类，它的对象是字符串变量。StringVar 类的对象用动态数组来实现，数组长度在程序运行时确定。因此，StringVar 类型的对象拥有动态数组的全部优点，同时还具有另一些功能。我们会定义 StringVar 的成员函数，确保一旦为 StringVar 对象赋的字符串太长，就显示错误消息。这里定义的版本只支持少量字符串对象操作。编程项目 1 要求增强类定义，添加更多的成员函数并对操作符进行重载。

由于可以使用标准 string 类(已在第 8 章讨论)，所以并不是真的需要这个 StringVar 类。但是，它的设计和编码过程对你而言是一个很好的练习。

图 11.11 给出了 StringVar 类型的定义。StringVar 类的一个构造函数要获取单个 int 实参。该实参决定了对象中存储的字符串值的最大长度。默认构造函数创建最大长度为 100 的对象。另一个构造函数获取数组实参，数组实参中包含一个 C 字符串(第 8 章讨论的那种)。注意，这意味着传给该构造函数的实参可以是引号字符串。该构造函数初始化一个对象，使其能容纳长度小于或等于其实参长度的任何字符串，并将对象的字符串值初始化成它的实参值的一个拷贝。目前暂时忽略“拷贝构造函数”。还要忽略名为~StringVar 的成员函数。虽然~StringVar 看起来像是构造函数，但实则不然。我们将在后续的小节里讨论它们。至于 StringVar 类的其他成员函数，它们的含义是很容易理解的。

图 11.11 给出了一个简单的示范程序。在 conversation 函数的定义中声明了两个对象，分别是 yourName 和 ourName。yourName 对象可包含任意字符串，只要它的长度小于或等于 maxNameSize。ourName 对象初始化成字符串值"Borg"，而且它的值可更改为长度小于或等于 4 的其他任何字符串。

图 11.11　使用 StringVar 类的程序

```
1   // 这是 StringVar 类的定义，用于表示字符串值
2   // 类的对象这样声明：StringVar theObject(maxSize);
3   // 注意是(maxSize)而不是[maxSize]。其中，maxSize 是允许的最长的字符串长度
4   #include <iostream>
5   using namespace std;
6
7
8       class StringVar
9       {
10      public:
11          StringVar(int size);
12          // 初始化对象，使其能够接受长度最大为 size 的字符串值
13          // 将对象的值设置成空字符串
14
15          StringVar();
16          // 初始化对象，使其能够接受长度最大为 100 的字符串值
17          // 将对象的值设置成空字符串
18
19          StringVar(const char a[]);
20          // 前条件：数组 a 包含以'\0'终止的一组字符
21          // 初始化对象，使它的值成为 a 中存储的字符串
22          // 并使其以后能设置成最大长度为 strlen(a)的字符串值
23
24          StringVar(const StringVar& stringObject);
25          // 拷贝构造函数(copy constructor)
```

```
26
27          ~StringVar();
28          // 将对象占用的所有动态内存返还给自由存储
29
30          int length() const;
31          // 返回当前字符串值的长度
32
33           void inputLine(istream& ins);
34          // 前条件：如果 ins 是一个文件输入流，那么 ins 已经连接到一个文件
35          // 行动：将输入流 ins 中直到'\n'的剩余文本复制到调用对象
36          // 如果没有足够空间，那么能复制多少就复制多少
37
38          friend ostream& operator <<(ostream& outs, const StringVar& theString);
39          // 重载<<操作符，使其能输出 StringVar 类型的值
40          // 前条件：如果 outs 是一个文件输出流，那么 outs 已经连接到一个文件
41
42      private:
43          char *value; // 指向容纳字符串值的动态数组的指针
44          int maxLength; // 任何字符串值的已声明最大长度
45      };
46
47
48       <成员函数和重载操作符的定义放在此处>
49
50  // 该程序演示 StringVar 类的使用
51
52  void conversation(int maxNameSize);
53  // 开始与用户的对话
54
55  int main()
56  {
57      using namespace std;
58      conversation(30);  <------------ 函数调用终止后，内存会还给自由存储
59      cout << "End of demonstration.\n";
60      return 0;
61  }
62
63  // 这只是一个演示性的函数：
64  void conversation(int maxNameSize)
65  {
66      using namespace std;
67                                       ┌------- 确定动态数组的长度
68                                       ↓
69      StringVar yourName(maxNameSize), ourName("Borg");
70
71      cout << "What is your name?\n";
72      yourName.inputLine(cin);
73      cout << "We are " << ourName << endl;
74      cout << "We will meet again " << yourName << endl;
75  }
```

示范对话

```
What is your name?
Kathryn Janeway
We are Borg
We will meet again Kathryn Janeway
End of demonstration.
```

如本小节开头所述，StringVar 类用动态数组实现。具体实现如图 11.12 所示。声明 StringVar 类型的对象时，会调用构造函数初始化对象。构造函数使用操作符 new 为成员变量 value 新建字符动态数组。字符串值在 value 数组中作为普通的字符串值来存储，并用'\0'标记字符串结束。注意，只有在声明对象时，才会确定这个数组的长度。那时会调用构造函数，而传给构造函数的实参决定了动态数组的长度。如图 11.11 所示，这个实参

可以是 int 变量。注意 conversation 函数的定义中 yourName 对象的声明。传给构造函数的实参是传值参数 maxNameSize。记住，传值参数是局部变量，所以 maxNameSize 是变量。任何 int 变量都可用这种方式传给构造函数。

成员函数 length、inputLine 和重载的输出操作符<<的实现都很容易明白。后续几个小节将讨论~StringVar 函数以及拷贝构造函数。

析构函数

视频讲解：*Arrays of Classes using Dynamic Arrays*

动态变量有一个问题。除非程序恰当地调用了 delete，否则它们不会“离去”。即便用局部指针变量创建动态变量，而且局部指针变量在函数调用结束时离去，除非调用 delete，否则动态变量仍会保留在内存中。不调用 delete 销毁动态变量，动态变量就会一直占用内存空间，这会导致程序因耗尽自由存储而终止。此外，将动态变量嵌入类的实现中，由于使用该类的程序员并不知道动态变量的存在，所以不能指望他们帮你调用 delete。事实上，由于数据成员通常是私有成员，所以程序员通常不能访问所需的指针变量，所以根本不能为这些指针变量调用 delete。为了解决这个问题，C++提供了称为析构函数的特殊成员函数。

析构函数(destructor)是成员函数，在类的对象离开作用域时自动调用。换言之，如函数包含局部变量，而且这个局部变量是提供了析构函数的对象，那么函数调用终止时会自动调用析构函数。如正确定义了析构函数，析构函数就会调用 delete 销毁由对象创建的所有动态变量。为达到“析构”之目的，可能只需要调用一次 delete，也可能需要调用多次。可让析构函数执行其他清理工作，但将内存回收到自由存储是析构函数的主职。

成员函数~StringVar 就是 StringVar 类(图 11.11)的析构函数。类似于构造函数，析构函数总是与定义它的类同名。但析构函数要在名称前添加符号~。符号~表明这是析构函数，而不是构造函数。与构造函数相似，析构函数不能指定返回值类型，甚至不能指定 void。构造函数是无参的。所以，每个类只能有一个析构函数，不能为类重载析构函数。除了这些区别，析构函数的定义方式与其他成员函数相同。

注意图 11.12 的析构函数~StringVar 的定义。析构函数~StringVar 调用 delete 销毁成员指针变量 value 指向的动态数组。再次分析图 11.11 的 conversation 函数。局部变量 yourName 和 ourName 都会创建动态数组。如果类没有析构函数，在对 conversation 的调用结束之后，动态数组仍会占用内存，即使它们对于程序来说已完全无用。对于这个示范程序，似乎不是很严重的问题，因为在对 conversation 的调用结束之后，程序会马上终止。但如果写程序反复调用 conversation 这样的函数，而且 StringVar 类没有合适的析构函数，函数调用就会不断消耗自由存储中的内存，直至所有内存都消耗殆尽，造成程序不得不异常终止。

图 11.12　StringVar 类的实现

```
1    // 这是 StringVar 类的实现;StringVar 类的定义如图 11.11 所示
2    #include <cstdlib>
3    #include <cstddef>
4    #include <cstring>
5
6     // 使用 cstddef 和 cstdlib:
```

```
7       StringVar::StringVar(int size) : maxLength(size)
8       {
9           value = new char[maxLength + 1]; //+1 是为'\0'留下的位置
10          value[0] = '\0';
11      }
12
13      // 使用 cstddef 和 cstdlib:
14      StringVar::StringVar() : maxLength(100)
15      {
16          value = new char[maxLength + 1]; // +1 是为'\0'留下的位置
17          value[0] = '\0';
18      }
19
20      // 使用 cstring，cstddef 和 cstdlib:
21      StringVar::StringVar(const char a[]) : maxLength(strlen(a))
22      {
23          value = new char[maxLength + 1]; // +1 是为'\0'留下的位置
24          strcpy(value, a);
25      }
26
27      // 使用 cstring，cstddef 和 cstdlib:
28      StringVar::StringVar(const StringVar& stringObject)    <—— 拷贝构造函数
29                          : maxLength(stringObject.length())     (本章稍后讨论)
30      {
31          value = new char[maxLength + 1]; // +1 是为'\0'留下的位置
32          strcpy(value, stringObject.value);
33      }
34       StringVar::~StringVar()    <—— 析构函数
35      {
36          delete [] value;
37      }
38
39      // 使用 cstring:
40      int StringVar::length() const
41      {
42          return strlen(value);
43      }
44
45      // 使用 iostream:
46      void StringVar::inputLine(istream& ins)
47      {
48          ins.getline(value, maxLength + 1);
49      }
50
51      // 使用 iostream:
52      ostream& operator <<(ostream& outs, const StringVar& theString)
53      {
54          outs << theString.value;
55          return outs;
56      }
```

析 构 函 数

析构函数(destructor)是类的成员函数，在类的对象离开作用域之后自动调用。例如，假定类类型的对象是某函数的局部变量，调用析构函数就是函数调用结束前采取的最后一项行动。析构函数销毁由对象创建的任何动态变量，将其占用的内存还给自由存储。析构函数还可执行其他清理工作。析构函数的名称必须由~符号和类名构成。

陷阱：指针作为传值参数

传值参数是指针类型，行为就变得很微妙，容易引起麻烦。以图 11.13 的函数调用为例。sneaky 函数中的 temp 参数是传值的，所以是局部变量。调用函数时，temp 的值被设置成参数 p 的值，并执行函数体。由于 temp 是局部变量，所以对 temp 的任何更改都只在

sneaky 内部生效。具体地说，指针变量 p 的值不应改变。但是，根据示范对话，指针变量 p 的值似乎已经改变。调用 sneaky 函数前*p 的值是 77，调用后是 99。为什么？

图 11.13　一个传值的指针参数

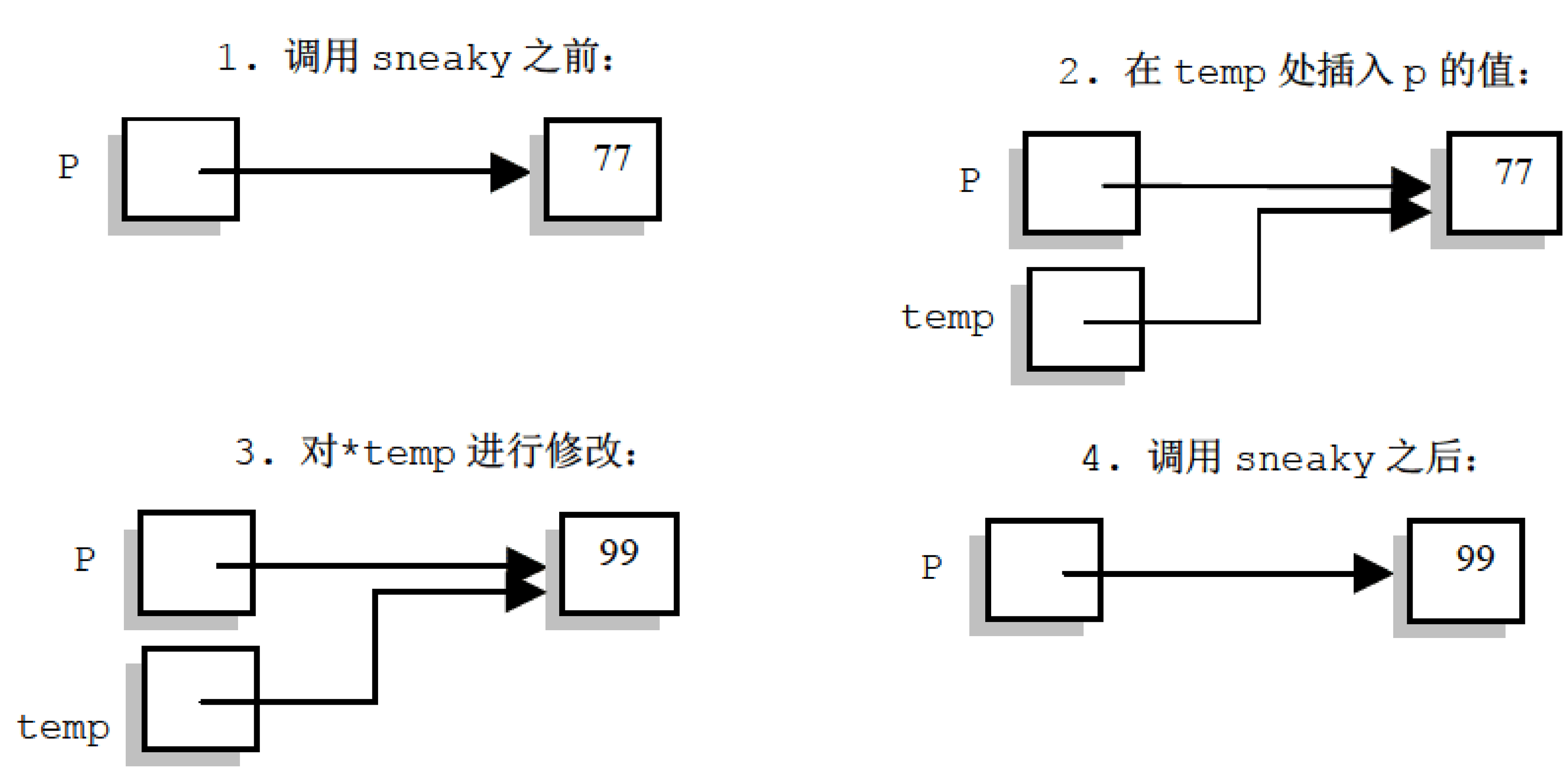
```
1   // 该程序演示了以传值的方式传递指针实参
2   #include <iostream>
3   using namespace std;
4
5   typedef int* IntPointer;
6
7   void sneaky(IntPointer temp);
8
9   int main()
10  {
11      IntPointer p;
12
13      p = new int;
14      *p = 77;
15      cout << "Before call to function *p == "
16           << *p << endl;
17      sneaky(p);
18      cout << "After call to function *p == "
19           << *p << endl;
20
21      return 0;
22  }
23
24  void sneaky(IntPointer temp)
25  {
26      *temp = 99;
27      cout << "Inside function call *temp == "
28           << *temp << endl;
29  }
30
```

示范对话

```
Before call to function *p == 77
Inside function call *temp == 99
After call to function *p == 99
```

图 11.14 对具体发生的事情进行了图示。虽然示范对话表明 p 似乎已经改变，但 p 的值实际没有被 sneaky 函数调用改变。指针 p 关联了两样东西：p 的指针值和 p 指向的值。现在说的“p 的值”是指针(内存地址)。调用 sneaky 后，变量 p 包含相同的指针值(相同的内存地址)。对 sneaky 的调用只是改变了 p 指向的变量值，p 本身的值(内存地址)并未改变。

图 11.14　sneaky(p);调用过程



如参数类型是类或结构，而且这个类或结构有一个指针类型的成员变量，那么以传值方式传递类类型的实参时，结果可能同样出人意料。然而，对于类类型，可以定义一个拷贝构造函数，避免(和控制)这种出人意料的改变，详情参见下一节。 ■

拷贝构造函数

拷贝构造函数(copy constructor)要求获取一个参数，该参数具有与类相同的类型。该参数必须传引用，而且通常要在前面附加 const 参数修饰符，使它成为常量参数。除此之外，拷贝构造函数的定义和用法与其他任何构造函数完全相同。

例如，在使用了图 11.11 的 StringVar 类的程序中，可能包含以下代码：

```
StringVar line(20), motto("Constructors can help.");
cout << "Enter a string of length 20 or less:\n";
line.inputLine(cin);
StringVar temp(line); // 由拷贝构造函数初始化
```

初始化 StringVar 类型的 3 个对象时，具体使用哪个构造函数取决于对象名后的圆括号中的实参类型。对 line 对象进行初始化的构造函数接收一个 int 类型的实参；对 motto 对象进行初始化的构造函数接收一个 const char a[]类型的实参。类似地，对 temp 对象进行初始化的构造函数接收一个 const StringVar&类型的实参。在这方面，拷贝构造函数与其他任何构造函数无异。

定义拷贝构造函数时，要确保初始化的对象成为函数实参的一个完整的、独立的拷贝。在以下声明中：

```
StringVar temp(line);
```

成员变量 temp.value 不能简单地设置成与 line.value 相同的值，那样会造成两个指针指向同一个动态数组。拷贝构造函数的定义如图 11.12 所示。注意，在定义中是新建了一个动态数组，并将一个动态数组的内容拷贝到另一个动态数组。所以，对于上述声明，temp 在初始化之后，它的字符串值等于 line 的字符串值，但 temp 有一个独立的动态数组。这样一来，对 temp 的任何更改都不会影响 line。这称为**深拷贝**。第 10 章讨论过浅拷贝，它只拷贝对内存中同一个动态数组的引用。深拷贝则生成任何动态结构的一个新拷贝。

如你所见，拷贝构造函数可以像其他任何构造函数那样使用。一些情况下，还会自动调用拷贝构造函数。大致地说，C++一旦需要某个对象的拷贝，就会自动调用拷贝构造函数。具体地说，拷贝构造函数会在三种情况下自动调用：①声明类的对象，并由同类型的另一个对象初始化；②函数返回类类型的值；③在传值形参的位置“插入”类类型的实参。在第三种情况下，拷贝构造函数定义了“插入”的具体含义。

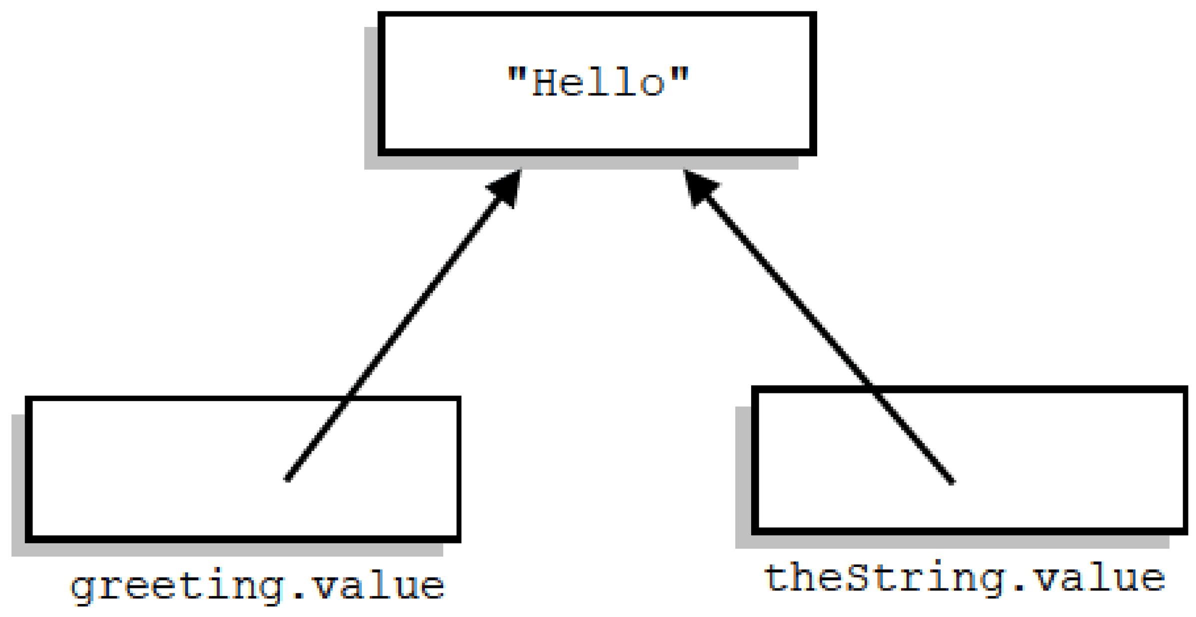
为了体会为什么需要拷贝构造函数，我们来看看不为 StringVar 类定义拷贝构造函数会发生什么。假设不在 StringVar 类的定义中包括拷贝构造函数，并假设函数定义中使用了一个传值参数，例如：

```
void showString(StringVar theString)
{
    cout << "The string is: "
         << theString << endl;
}
```

再给定以下代码，其中包括一个函数调用：

```
StringVar greeting("Hello");
showString(greeting);
cout << "After call: " << greeting << endl;
```

假定没有拷贝构造函数，那么具体过程是：执行函数调用时，greeting 的值复制给局部变量 theString，所以 theString.value 被设置成与 greeting.value 相等。但是，这些是指针变量，所以在函数调用期间，theString.value 和 greeting.value 指向同一个动态数组(如下所示)。



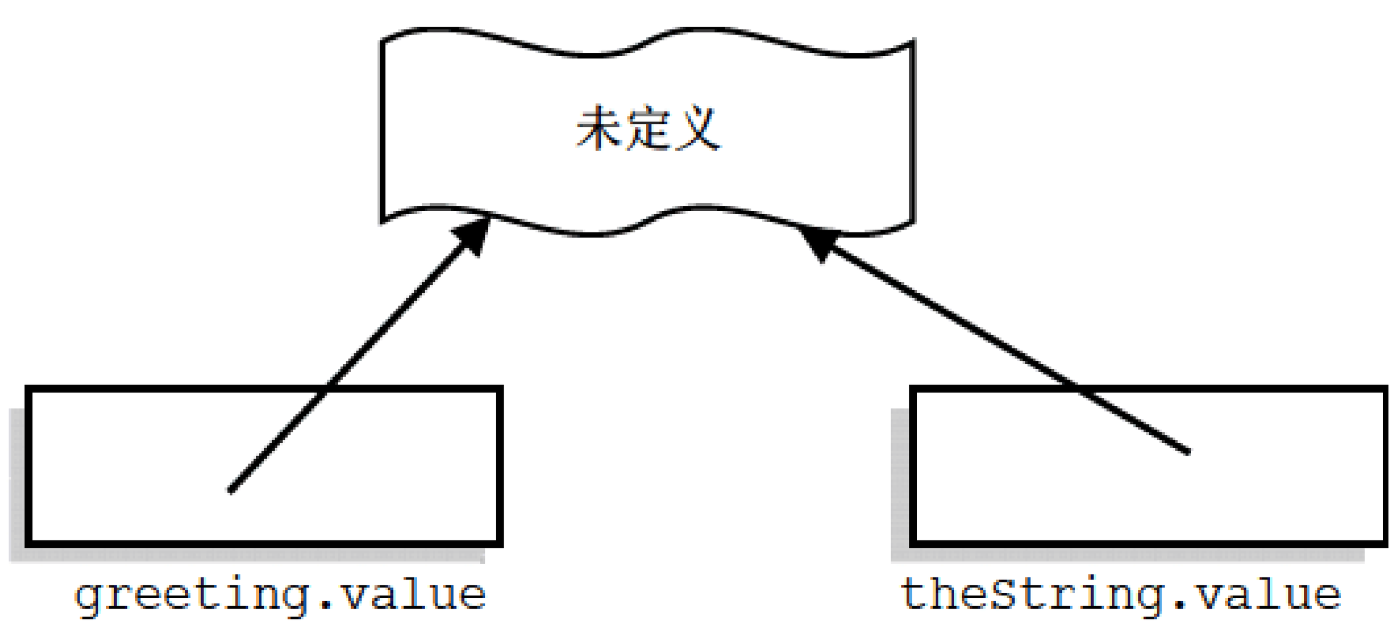
函数调用结束之后，会调用 StringVar 的析构函数，将 theString 使用的内存返还给自由存储。析构函数的定义包括以下语句：

```
delete [] value;
```

由于析构函数是为 theString 对象调用的，所以上述语句相当于：

```
delete [] theString.value;
```

所以以上图示变成下面这样：



由于 greeting.value 和 theString.value 指向同一个动态数组，所以删除 theString.value 就是删除 greeting.value。一旦程序抵达以下语句，greeting.value 就会进入未定义状态：

```
cout << "After call: " << greeting << endl;
```

所以，这个 cout 语句是未定义的。个别情况下，cout 语句也许能提供你希望的输出，但“greeting.value 未定义”这一事实迟早会出问题。如果 greeting 对象是某些函数的局部变量，就会出重大问题。在这种情况下，析构函数调用等价于：

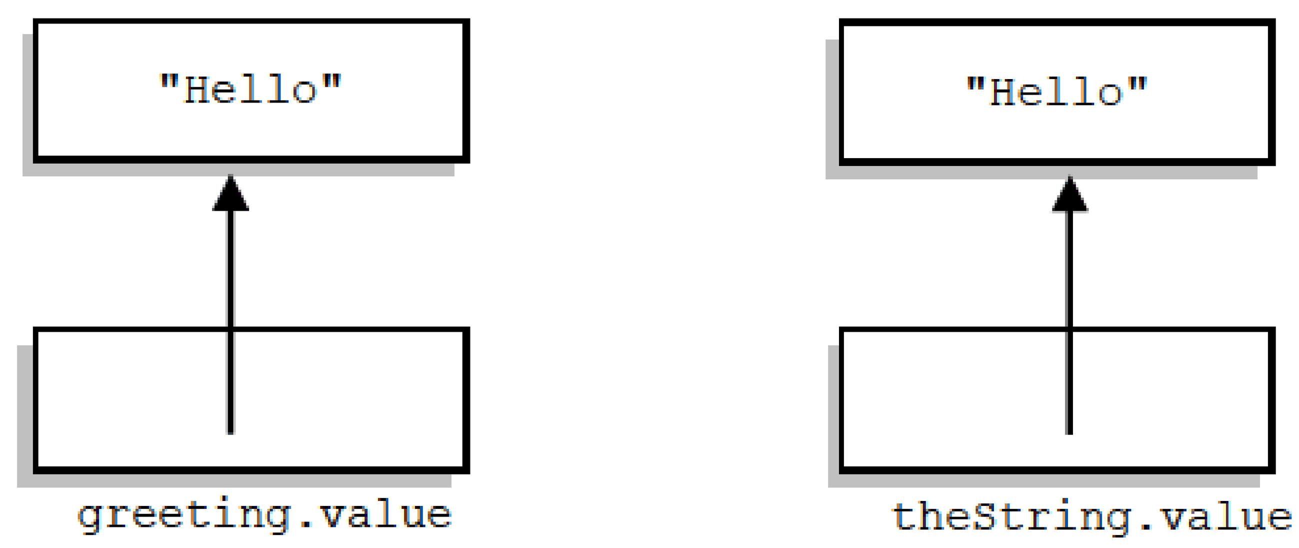
```
delete [] greeting.value;
```

但是，就像刚才说的那样，greeting.value 指向的动态数组已被删除一次，现在系统试图再把它删除一次。重复调用 delete 来删除同一个动态数组(或者用 new 创建的其他变量)可能造成严重的系统错误，并导致程序崩溃。

如果没有拷贝构造函数，就会发生这种事情。幸好，我们在 StringVar 类的定义中包括了一个拷贝构造函数，所以在执行以下函数调用时，会自动调用拷贝构造函数：

```
StringVar greeting("Hello");
showString(greeting);
```

拷贝构造函数定义了什么叫做在传值形参 theString 的位置“插入”实参 greeting，前面的图示变成下面的形式：



因此，对 theString.value 进行的任何修改都不会影响实参 greeting，析构函数也不会再出问题。如果为 theString 调用析构函数，再为 greeting 调用析构函数，每次调用都会销毁一个不同的动态数组。

函数返回类类型的值时，会自动调用拷贝构造函数，以复制由返回语句指定的值。如果没有拷贝构造函数，会产生与传值形参相似的问题。

如果类定义涉及指针以及使用操作符 new 动态分配的内存，就需要包括一个拷贝构造函数。拷贝构造函数一般要对动态内存结构执行一次深拷贝。不涉及指针或动态分配内存的类不需要拷贝构造函数。默认执行的浅拷贝通常就足够了。

可能和你预想的不同，使用赋值操作符设置一个对象等于另一个对象时，不会调用拷贝构造函数。[①]然而，如果不喜欢默认赋值操作符的行为，可以重新定义赋值操作符，详情参见下一节。

拷贝构造函数

拷贝构造函数获取一个传引用参数，该参数具有与类相同的类型。参数必须传引用。通常，该参数也是常量参数，要在它前面附加参数修饰符 const。只要某个函数返回类类型的值，就会自动调用那个类的拷贝构造函数。在类类型的传值参数位置“插入”实参时，也会自动调用拷贝构造函数。拷贝构造函数还可采取与其他构造函数相同的方式来使用。

凡是使用了指针和操作符 new 的类，都应该有一个拷贝构造函数。

Big Three

拷贝构造函数、操作符=以及析构函数统称为 Big Three。专家认为，如果需要定义其中一个，就必须定义全部三个。缺少任何一个，编译器都会帮你创建它，只是可能达不到你预期的效果。所以，有必要每次都自己定义。假如所有成员变量都具有预定义类型(比如 int 和 double)，那么编译器为你生成的拷贝构造函数和重载的操作符=能很好地工作。但假如类中包含类或指针成员变量，它们就可能表现失常。对于使用了指针和操作符 new 的任何类，最保险的做法是定义自己的拷贝构造函数、重载的操作符=以及析构函数。

① C++对初始化(要调用拷贝构造函数的 3 种情况)和赋值进行了细致区分。初始化是用拷贝构造函数新建一个对象；赋值操作符则获取一个现有的对象，并对其进行修改，使其成为与赋值操作符右侧的对象完全相同的一个拷贝(两者除了在内存中的位置不同，其他完全相同)。

自测题

23. 如果一个类名为 MyClass，而且它有一个构造函数，那么这个构造函数的名称是什么？如果 MyClass 还有一个析构函数，析构函数的名称又是什么？

24. 如果像下面这样修改图 11.12 的析构函数定义，图 11.11 的示范对话会发生什么变化？

```
StringVar::~StringVar()
{
    cout << endl
         << "Good-bye cruel world! The short life of\n"
         << "this dynamic array is about to end.\n";
    delete [] value;
}
```

25. 下面是 StringVar 类的拷贝构造函数定义的第一行。标识符 StringVar 出现了 3 次，意味着每次都稍有区别。在这 3 种情况下，StringVar 的含义是什么？

```
StringVar::StringVar(const StringVar& stringObject)
```

26. 回答以下有关析构函数的问题。

   a. 什么是析构函数，析构函数的命名有何要求？
   b. 什么时候会调用析构函数？
   c. 析构函数实际能做哪些事情？
   d. 析构函数的设计用途是什么？

重载赋值操作符

视频讲解：*Overloading = and == for a Class*

假定 string1 和 string2 像下面这样声明：

```
StringVar string1(10), string2(20);
```

StringVar 类已在图 11.11 和图 11.12 定义。假定 string2 已赋值，以下赋值语句虽然合法，但含义可能和你想的不同：

```
string1 = string2;
```

和往常一样，这个预定义版本的赋值语句将 string2 的每个成员变量的值复制到 string1 的对应成员变量中。这样，string1.maxLength 的值就变成与 string2.maxLength 的值相同，string1.value 变成与 string2.value 相同。但这可能造成 string1 出问题，甚至 string2 都可能出问题。

成员变量 string1.value 包含一个指针，赋值语句将该指针设为与 string2.value 相同的值。所以，string1.value 和 string2.value 现在指向内存中的同一个位置。改变 string1 中的字符串值，必然改变 string2 中的字符串值。改变 string2 中的字符串值，也必然改变 string1 中的字符串值。

总之，预定义赋值语句不能做我们想为 StringVar 对象做的事情。为 StringVar 类使用预定义赋值操作符不会有什么好事。解决方案是重载赋值操作符=，使它能做我们想为 StringVar 对象做的事情。

赋值操作符不能像其他操作符(比如<<和+)那样重载。要重载赋值操作符，该操作符必须是类的成员，而不能是类的友元。下面修改 StringVar 类的定义来添加赋值操作符的重

载版本：

```
class StringVar
{
public:
    void operator =(const StringVar& rightSide);
    // 重载赋值操作符=，将字符串从一个对象复制到另一个
    <类剩余的定义与图 11.11 相同>
```

然后，重载的赋值操作符可以像普通赋值操作符那样使用。例如以下语句：

```
string1 = string2;
```

在上述调用中，string1 是调用对象，string2 是传给成员操作符=的实参。

赋值操作符像下面这样定义：

```
// 以下定义是可以接受的，但以后会给出一个更好的定义：
void StringVar::operator =(const StringVar& rightSide)
{
    int newLength = strlen(rightSide.value);
    if ((newLength) > maxLength)
        newLength = maxLength;

    for (int i = 0; i < newLength; i++)
        value[i] = rightSide.value[i];
    value[newLength] = '\0';
}
```

注意代码会检查赋值操作符右侧对象中的字符串长度。假如太长以至于赋值操作符左侧的对象(也就是调用对象)不能完全装下，那么能复制多少就复制多少。如果不想在复制过程中损失任何字符，为了填充所有字符，需要为赋值操作符左侧的对象新建一个更大的动态数组。为此，可像下面这样重新定义赋值操作符：

```
// 这个版本有一个 bug：
void StringVar::operator =(const StringVar& rightSide)
{
    delete [] value;
    int newLength = strlen(rightSide.value);
    maxLength = newLength;
    value = new char[maxLength + 1];
    for (int i = 0; i < newLength; i++)
         value[i] = rightSide.value[i];
    value[newLength] = '\0';
}
```

赋值操作符两侧是同一个对象，上述版本就会出问题，例如：

```
myString  = myString;
```

执行该赋值语句时，执行的第一个语句如下：

```
delete [] value;
```

调用对象是 myString，所以上述语句的等价形式如下：

```
delete [] myString.value;
```

因此，myString 中的字符串值会被删除，造成指针 myString.value 进入未定义状态。赋值操作符破坏了 myString 对象，像这样运行的程序可能崩溃。

要修正该 bug，一个办法是首先检查赋值操作符左侧对象的动态数组成员是否有足够大的空间，只有在需要额外空间时才删除数组。因此，在最终版本的重载赋值操作符定义中，我们新增了这个检查：

```
// 这才是最终版本:
void StringVar::operator =(const StringVar& rightSide)
{
    int newLength = strlen(rightSide.value);
    if (newLength > maxLength)
    {
        delete [] value;
        maxLength = newLength;
        value = new char[maxLength + 1];
    }
    for (int i = 0; i < newLength; i++)
        value[i] = rightSide.value[i];
    value[newLength] = '\0';
}
```

对于许多类，如果位于赋值操作符两侧的是同一个对象，重载赋值操作符时一定要小心。一定要检查这种情况，并仔细定义重载的赋值操作符，使其在这种特殊情况下也能正常工作。

√ 自测题

27. a. 在数据只由内建(预定义)类型构成的情况下，为什么不需要重载赋值操作符？请说明原因。
    b. 如果将(a)的赋值操作符更换为拷贝构造函数呢？
    c. 如果将(a)的赋值操作符更换为析构函数呢？

小　结

- 类的友元函数其实就是普通函数，只是能像成员函数那样访问类的私有成员。
- 如果类有一套完整的取值函数和赋值函数，将函数变成友元的唯一理由就是简化函数定义，并提高它的效率——但这些理由通常就足够了。
- 对于不被函数更改的类类型参数，通常应将其设为常量参数。
- 操作符(比如+和==)可以重载，使它们能应用于你定义的类类型的对象。
- 重载操作符>>或<<时，返回的类型应该是一个流类型，而且返回的类型必须是引用(为返回类型名称附加后缀&)。
- 数组基类型可以是结构或类类型。结构或类可将数组作为自己的成员变量。
- 析构函数是类的一种特殊成员函数，在类的对象离开作用域时自动调用。析构函数的主要使命是将内存还给自由存储，使内存得以重用。
- 拷贝构造函数是一种特殊构造函数，它有一个参数，该参数具有与类相同的类型。定义了拷贝构造函数之后，只要函数返回类类型的值，或者在类类型的一个传值参数位置“插入”一个实参，就会自动调用这个构造函数。使用了指针和操作符 new 的任何类都应包含一个拷贝构造函数。
- 可为一个类重载赋值操作符=，使它具有你希望的行为。但它必须作为类的成员来重载，不能作为友元。使用了指针和 new 操作符的任何类都应重载作用于那个类的赋值操作符。

自测题答案

1.
```
bool before(DayOfYear date1, DayOfYear date2)
{
    return( (date1.getMonth() < date2.getMonth())
          ||(date1.getMonth() == date2.getMonth()
              && date1.getDay() < date2.getDay()) );
}
```

上述布尔表达式表明：date1 要想在 date2 之前，那么 date1 的月份必须在 date2 的月份之前。如月份相同，则 date1 的天数必须在 date2 的天数之前。

2. 友元函数和成员函数的相似之处在于，它们都能在其函数定义中使用类的任何成员(无论公共还是私有成员)。但友元函数的定义和使用和普通函数相同。调用友元函数时不需要使用圆点操作符；定义友元函数时，不需要使用类型限定符。相反，成员函数要使用一个对象名和一个圆点操作符来调用。另外，成员函数的定义中包括一个类型限定符，它由类名和作用域解析操作符::构成。

3. 修改过的 DayOfYear 类定义如下所示。加粗部分是新增的。省略了部分注释以节省篇幅，但图 11.2 的所有注释都应包括在这个定义中。

```
class DayOfYear
{
public:
    friend bool equal(DayOfYear date1, DayOfYear date2);
    friend bool after(DayOfYear date1, DayOfYear date2);
    // 前条件：date1 和 date2 都已赋值
    // 如果 date1 在 date2 之后，就返回 true，否则返回 false

    DayOfYear(int theMonth, int theDay);
    DayOfYear();
    void input();
    void output();
    int getMonth();
    int getDay();
private:
    void checkDate();
    int month;
    int day;
};
```

还必须为 after 函数添加以下定义：

```
bool after(DayOfYear date1, DayOfYear date2)
{
    return  ((date1.month > date2.month)) ||
            ((date1.month == date2.month) && (date1.day > date2.day));
}
```

4. 修改过的 Money 类的定义如下所示。有底纹的部分是新增的。省略了部分注释以节省篇幅，但图 11.3 的所有注释都应包括在这个定义中。

```
class Money
{
public:
    friend Money subtract(Money amount1, Money amount2);
    // 前条件：amount1 和 amount2 都已赋值
    // 返回 amount1 减去 amount2 的结果

    friend bool equal(Money amount1, Money amount2);
    Money(long dollars, int cents);
    Money(long dollars);
    Money();
    double getValue();
    void input(istream& ins);
```

```
    void output(ostream& outs);
private:
    long allCents;
};
```

还必须为 subtract 函数添加以下定义：

```
Money subtract(Money amount1, Money amount2)
{
    Money temp;
    temp.allCents = amount1.allCents
                    - amount2.allCents;
    return temp;
}
```

5. 修改过的 Money 类的定义如下所示。有底纹的部分是新增的。省略了部分注释以节省篇幅，但图 11.3 的所有注释都应包括在这个定义中。

```
class Money
{
public:
    friend Money add(Money amount1, Money amount2);
    friend bool equal(Money amount1, Money amount2);
    Money(long dollars, int cents);
    Money(long dollars);
    Money();
    double getValue();
    void input(istream& ins);

    void output(ostream& outs);
    // 前条件：如果 outs 是一个文件输出流，那么 outs 已经连接到一个文件
    // 后条件：调用对象中记录好的美元符号和金额将发送到输出流 outs

    void output();
    // 后条件：调用对象中记录的一个美元符号和金额输出到屏幕
private:
    long allCents;
};
```

此外还必须为 output 函数添加以下定义(output 的原始定义仍然保留，所以会有 output 的两个定义)。

```
void Money::output()
{
    output(cout);
}
```

也可使用下面这个较长的函数定义版本：

```
// 使用 cstdlib 和 iostream
void Money::output()
{
    long positiveCents, dollars, cents;
    positiveCents = labs(allCents);
    dollars = positiveCents/100;
    cents = positiveCents%100;

    if (allCents < 0)
        outs << "-$" << dollars << '.';
    else
        outs << "$" << dollars << '.';
    if (cents < 10)
        outs << '0';
    cout << cents;
}
```

还可以重载成员函数 input，使以下调用：

```
purse.input();
```

等价于以下调用：

```
purse.input(cin);
```

当然还可合并上述所有修改，生成一个有较大幅度改进的 Money 类。

6. 如输入**$-9.95**(而不是**-$9.95**)，input 函数会读入'$'，并把它作为 oneChar 的值；读入-9，作为 dollars 的值；读入'.'，作为 decimalPoint 的值；并读入'9'和'5'，分别作为 digit1 和 digit2 的值。这意味着它会将 dollars 设为-9，而 cents 等于 95。因此，最终金额是-$9.00 加 0.95，结果是-$8.05。为了捕捉这个问题，一个办法是测试 dollars 的值是否为负(因为 dollars 的值应该是一个绝对值)。为此，请像下面这样重写错误消息部分：

```
if ( oneChar != '$' || decimalPoint != '.'
     || !isdigit(digit1) || !isdigit(digit2)
     || dollars < 0 ) ◄──────────────── 新增
{
    cout << "Error illegal form for money input\n";
    exit(1);
}
```

但我们的程序仍然检测不到美元数额为零的一个不正确的输入(比如$-0.95)。运用前面学到的知识对这种情况进行检测，虽然肯定可行，但不可避免地会使代码严重复杂化，进而影响可读性。

7.
```
#include <iostream>
using namespace std;
int main()
{
    int x;
    cin >> x;
    cout << x << endl;
    return 0;
}
```

如编译器将具有前置 0 的输入解释成八进制数字，那么输入 077，输出应该是 63。如编译器不将具有前置 0 的数据解释成八进制，那么输出应该是 77。

8. 对图 11.3 的版本进行的唯一修改就是在函数头中添加了修饰符 const，所以定义如下：

```
double Money::getValue() const
{
    return (allCents * 0.01);
}
```

9. 成员函数 input 更改它的调用对象的值，所以添加修饰符 const 编译器会报错。

10. 相同点：两种参数调用方法都能防止调用者的实参发生更改。不同点：传值的参数会生成调用者的实参的一个拷贝，所以相较于传 const 引用的参数，前者要使用更多的内存空间。

11. 在 const int x = 17;声明中，关键字 const 向编译器承诺：作者写的代码不会改变 x 的值。

    在 int f() const;声明中，关键字 const 向编译器承诺：作者为实现 f 函数而写的代码不会改变调用对象中的任何东西。

    在 int g(const A& x);声明中，关键字 const 向编译器承诺：由类作者写的代码不会更改用于取代 x 的实参值。

12. 包括+，*和/在内的(二元)操作符与函数的区别在于它们的调用语法。在函数调用中，参数值要放在函数名称之后的圆括号中。使用操作符时，参数值则放在操作符前后。另外，在重载的操作符的声明和定义中，都必须使用保留字 operator。

13. 修改过的 Money 类的定义如下所示。粗体部分是新增的。省略了部分注释以节省篇幅，但图 11.5 的所有注释都应包括在这个定义中。

```
class Money
{
public:
```

```
    friend Money operator +(const Money& amount1, const Money& amount2);
    friend bool operator ==(const Money& amount1, const Money& amount2);

    friend bool operator <(const Money& amount1, const Money& amount2);
    // 前条件：amount1 和 amount2 已被赋值
    // 如果 amount1 小于 amount2，就返回 true；否则返回 false

    Money(long dollars, int cents);
    Money(long dollars);
    Money();
    double getValue() const;
    void input(istream& ins);
    void output(ostream& outs) const;
private:
    long allCents;
};
```

还必须包括重载操作符<的定义，如下所示：

```
bool operator <(const Money& amount1, const Money& amount2)
{
    return (amount1.allCents < amount2.allCents);
}
```

14. 修改过的 Money 类的定义如下所示。粗体部分是新增的。省略了部分注释以节省篇幅，但图 11.5 的所有注释都应包括在这个定义中。其中已包括了来自上一个自测题的更改，这是因为在重载操作符<=的定义中，使用重载操作符<是一件很自然的事情。

```
class Money
{
public:
    friend Money operator +(const Money& amount1, const Money& amount2);
    friend bool operator ==(const Money& amount1, const Money& amount2);

    friend bool operator <(const Money& amount1, const Money& amount2);
    // 前条件：amount1 和 amount2 已被赋值
    // 如果 amount1 小于 amount2，就返回 true；否则返回 false

    friend bool operator <=(const Money& amount1, const Money& amount2);
    // 前条件：amount1 和 amount2 已被赋值
    // 如果 amount1 小于或等于 amount2，就返回 true；否则返回 false

    Money(long dollars, int cents);
    Money(long dollars);
    Money();
    double getValue() const;
    void input(istream& ins);
    void output(ostream& outs) const;
private:
    long allCents;
};
```

还必须包括重载操作符<=的定义(以及上一个自测题的重载操作符<的定义)，如下所示：

```
bool operator <=(const Money& amount1, const Money& amount2)
{
    return ((amount1.allCents < amount2.allCents)
           || (amount1.allCents == amount2.allCents));
}
```

15. 如重载操作符，操作符的至少一个实参必须是类类型，所以+之于整数的行为无法改变。事实上，正是因为存在这个要求，任何操作符之于任何内建类型的行为都无法改变。

16.
```
// 使用 cmath(为了使用其中的 floor)：
Money::Money(double amount)
{
    allCents = floor(amount * 100);
}
```

该定义会丢弃小于 1 美分的任何金额。例如，它将 12.34999 转换成整数 1234，代表金额$12.34。也可定义构造函数，让它对小于一分钱的零头进行其他处理。

17.
```
istream& operator>>(istream& ins, Pairs& second)
{
    char ch;
    ins >> ch;        // 丢弃开头的'('
    ins >> second.f;
    ins >> ch;        // 丢弃逗号','
    ins >> second.s;
    ins >> ch;        // 丢弃最后的')'
    return ins;
}
ostream& operator<<(ostream& outs, const Pairs& second))
{
    outs << '(';
    outs << second.f;
    outs << ','; // 也可选择', '，获得一个额外的空格
    outs << second.s;
    outs << ')';
    return outs;
}
```

18.
```
// 使用 iostream:
istream& operator >>(istream& ins, Percent& theObject)
{
    char percentSign;
    ins >> theObject.value;
    ins >> percentSign; // 丢弃符号%
    return ins;
}

// 使用 iostream:
ostream& operator <<(ostream& outs, const Percent& aPercent)
{
    outs << aPercent.value << '%';
    return outs;
}
```

19.
```
struct Score
{
    int homeTeam;
    int opponent;
};
Score game[10];
```

20.
```
// 读入 5 个金额，倍增每个金额，然后输出结果
#include <iostream>
#include "money.h"

< Money 类定义放在这里>

int main()
{
    using namespace std;
    Money amount[5];
    int i;
    cout << "Enter 5 amounts of money:\n";
    for (i = 0; i < 5; i++)
        cin >> amount[i];
    for (i = 0; i < 5; i++)
    amount[i] = amount[i] + amount[i];
    cout << "After doubling, the amounts are:\n";
    for (i = 0; i < 5; i++)
        cout << amount[i] << " ";
    cout << endl;

    return 0;
}
```

(不能使用 2*amount[i]，因为没有为 Money 类型的操作数重载*操作符。)

21. 参见自测题 22 的答案。

22. 下面合成了本题和上一题的答案。对类定义的修改如下所示。为节省篇幅，删除了图 11.10 的部分注释，但在你自己的答案中应包括这些注释。

```
class TemperatureList
{
public:
    TemperatureList();

    int getSize() const;
    // 返回列表中的温度数量

    void addTemperature(double temperature);

    double getTemperature(int position) const;
    // 前条件：0 <= position < getSize()
    // 返回在指定位置添加的温度。添加的第一个温度在位置 0

    bool full() const;

    friend ostream& operator <<(ostream& outs, const TemperatureList& theObject);

private:
     double list[MAX_LIST_SIZE]; // list 数组存储的是华氏温度
     int size; // 实际填充的数组位置数目
};
```

还需添加以下成员函数定义：

```
int TemperatureList::getSize() const
{
    return size;
}

// 使用 iostream 和 cstdlib:
double TemperatureList::getTemperature (int position) const
{
    if ( (position >= size) || (position < 0) )
    {
        cout << "Error:"
             << " reading an empty list position.\n";
        exit(1);
    }
    else
    {
        return (list[position]);
    }
}
```

23. 构造函数名为 MyClass，与类同名。析构函数名为~MyClass。

24. 对话会变成下面这样：

```
What is your name?
Kathryn Janeway
We are Borg
We will meet again Kathryn Janeway
Good-bye cruel world! The short life of
this dynamic array is about to end.
Good-bye cruel world! The short life of
this dynamic array is about to end.
End of demonstration
```

25. 位于::左侧的 StringVar 是类名。位于::右侧的 StringVar 是成员函数名称(记住，构造函数本质上是一个与类同名的成员函数)。圆括号中的 StringVar 是 stringObject 参数的类型。

26. a. 析构函数是类的一个成员函数。析构函数的名称总是以~开头，后跟类名。
    b. 析构函数在类的一个对象离开作用域时自动调用。
    c. 析构函数实际能做程序员要求它做的任何事情！
    d. 析构函数设计用来删除由构造函数为类的对象分配的动态变量。析构函数还能做其他清理工作。

27. 在赋值操作符=和拷贝构造函数的情况下，如数据全都是内建类型，那么默认的拷贝机制正是我们需要的。在析构函数的情况下，由于本来就没有分配动态内存(没有指针)，所以默认的“什么都不做”恰好是我们希望的行为。

编 程 练 习

编程练习一般只需写很小的程序，运用本章提到的编程概念。

1. 修改图 11.8 的 Money 类定义，添加以下所有内容。

   a. 重载操作符<，<=，>和>=，使它们支持 Money 类型(**提示**：参考自测题 13)。
   b. 将以下成员函数添加到类定义(显示的是类定义中的函数声明。函数定义本身将包括限定符 Money::)：

   ```
   Money percent(int percentFigure) const;
   // 返回调用对象中的金额的一个百分比
   // 例如，假定 percentFigure 是 10，那么返回值是调用对象所表示的金额的 10%
   ```

   例如，假定 purse 是 Money 类型的一个对象，它的值表示金额$100.10，那么以下调用：

   ```
   purse.percent(10);
   ```

   将返回$100.10 的 10%。换言之，它会返回 Money 类型的一个值，该值表示金额$10.01。

2. 自测题 17 要求为 Pairs 类重载操作符>>和操作符<<。请完成那道题并进行测试。实现默认构造函数，再实现接收一个和两个 int 参数的构造函数。接收一个参数的构造函数应该初始化数偶的第一个成员；第二个成员则应初始化为 0。

   重载的二元操作符+要按照以下规则来实现数偶相加：

   ```
   (a, b) + (c, d) = (a + c, b + d)
   ```

   以类似的方式重载 operator-。

   按以下规则重载应用于 Pairs 和 int 类型的 operator*：

   ```
   (a, b) * c = (a * c, b * c)
   ```

   写程序来测试类定义中的所有成员函数和重载操作符。

3. 自测题 18 要求为 Percent 类重载操作符>>和<<。请完成并测试这道题。实现默认构造函数，然后实现接收一个 int 参数的构造函数。重载操作符+和-，实现百分比的加法和减法。另外，重载操作符*，允许百分比乘以一个整数。

   写程序测试类定义中的所有成员函数和重载的操作符。

编 程 项 目

编程项目要求综合运用多方面的知识来解决问题，程序一般比编程练习大，解题方式多样化。

1. 第 8 章讨论了向量，它与数组相似，区别在于长度能自由变化。假定 C++没有定义向量。请定义一个名为 VectorDouble 的类，它类似于基类型为 double 的向量。VectorDouble 类有一个私有成员变量，它是一个 double 类型的动态数组。还有两个 int 类型的成员变量：一个是 maxCount，代表 double 类型的动态数组的长度；另一个是 count，代表目前实际存储了值的数组位置数目(maxCount 相当于向量的容量，count 相当于向量的长度)。

如果为 VectorDouble 类型的向量对象添加一个元素(一个 double 值)，但没有更多的空间，就会新建一个动态数组，它的容量是以前创建的旧动态数组的两倍，而且旧数组的值会复制给新数组。

你的类应该包含以下内容。

- 3 个构造函数：一个默认构造函数，创建有 50 个元素的一个动态数组；一个要获取一个 int 参数的构造函数，该参数指定了初始动态数组中的元素数目；一个拷贝构造函数。
- 一个析构函数。
- 赋值操作符=的一个合适的重载版本。
- 相等性操作符==的一个合适的重载版本。若要相等，count 以及数组元素的值必须相等，但 maxCount 不一定相等。
- 成员函数 push_back，capacity，size，reserve 和 resize，行为和向量的同名成员函数一样。
- 两个成员函数，为你的类赋予与方括号相同的功能：valueAt(i)，返回动态数组的编号为 i 的元素的值。以及 changeValueAt(d, i)，它将动态数组的编号为 i 的元素的 double 值更改为 d。必须对传给 valueAt 和 changeValueAt 的实参进行适当的限制(类还不支持方括号。支持方括号还需学习更多知识)。

2. 定义有理数类。有理数表示成两个相除的整数，比如 1/2，3/4，64/2 等(对于 1/2 这样的表达式，我们指的是普通的分数，不要把它们想象成在 C++程序中会造成整数除法的表达式)。将有理数表示成 int 类型的两个值：一个表示分子，一个表示分母。将类命名为 Rational。

   先包括一个接收两个参数的构造函数，该构造函数可将一个对象的成员变量设为任何合法的值。然后包括一个接收一个 int 参数的构造函数，将这个单独的参数命名为 wholeNumber，并定义构造函数，使对象初始化成有理数 wholeNumber/1。再包括一个默认构造函数，将对象初始化成 0(也就是 0/1)。

   重载输入和输出操作符>>和<<。数字以 1/2，15/32，300/401 这样的形式来输入和输出。注意分子、分母可能包含减号，所以必须支持像-1/2，15/-32 以及-300/-401 这样的输入。重载以下所有操作符，使它们能正确支持 Rational 类型：==，<，<=，>，>=，+，-，*和/。写测试程序测试该类。

   提示：如果 a*d 等于 c*b，那么有理数 a/b 和 c/d 是相等的。如果 b 和 d 是正的有理数，那么只要 a*d 小于 c*b，a/b 就肯定小于 c/d。你应该包括一个函数，让它对存储的值进行正规化。正规化之后，分母肯定是正的，而且分子和分母肯定是最小的。例如，对 4/-8 正规化之后，它应该表示成等价的-1/2。

3. 定义复数类。复数的形式如下：

   ```
   a + b*i
   ```

   其中，a 和 b 是 double 类型的数字，而 i 是表示虚数单位 $\sqrt{-1}$ 的一个数字 。将复数表示成 double 类型的两个值。将成员变量命名为 real 和 imaginary(要用 i 来乘的那个数字的变量就是 imaginary，也就是虚数)。将这个类命名为 Complex。

   请包括接收两个 double 类型的变量作为参数的一个构造函数，它用于将一个对象的成员变量设为任意值。还要包括只接收单个 double 类型的变量作为参数的一个构造函数；将这个参数命名为 realPart，并定义构造函数，使对象初始化为 realPart + 0*i。再包括一个默认构造函数将对象初始化为 0(也就是 0 + 0*i)。重载以下所有操作符，使它们都能正确地支持 Complex 类型：==，+，-，*，>>和<<。写测试程序测试该类。

   提示：要实现两个复数的加法和减法，可以加减 double 类型的两个成员变量。要计算两个复数的乘积，请使用以下公式：

   $$(a + b*i)*(c + d*i) == (a*c - b*d) + (a*d + b*c)*i$$

   在接口文件中，应该像下面这样定义常量 i：

   ```
   const Complex i(0, 1);
   ```

   这个定义的常量 i 等同于前面讨论的 i。

4. 改进图 11.11 和图 11.12 给出的 StringVar 类定义，添加以下内容。

- 成员函数 copyPiece，返回一个指定子字符串；成员函数 oneChar，返回指定的单个字符；成员函数 setChar，更改一个指定字符。
- 操作符==的一个重载版本(注意，只要求字符串值相等；maxLength 的值不要求相等)。
- 操作符+的一个重载版本，用于连接 StringVar 类型的字符串。
- 提取操作符>>的一个重载版本，读入一个单词(inputLine 则相反，它读入整行)。

如果学完了 11.4 节，请将重载的赋值操作符也加入这个改进版本中。另外，写合适的测试程序全面测试类定义。

5. 定义 Text 类，其对象用于存储单词列表。Text 类与 StringVar 类相似，只是使用基类型为 StringVar(而非 char)的一个动态数组，并使用由单独一个空格构成的 StringVar 对象来标记数组的结束，而不是使用'\0'作为结束标记。直观地看，Text 类的对象是由一组单词构成的文本，每个单词以空格分隔。要对 StringVar 类型的数组元素进行限制：除了 StringVar 类型的结束标记元素之外，其他元素都不包含空格。

   Text 类提供和 StringVar 类对应的成员函数。一个构造函数获取 const char a[]类型的实参，采用与 inputLine 相同的方式初始化 Text 对象，具体请参见后文对 inputLine 的描述。如果 C 字符串实参包含换行符'\n'，就认为出错，显示错误消息并终止程序。

   成员函数 inputLine 读取以空格分隔的字符串，将字符串存储到基类型为 StringVar 的动态数组的元素中。连续多个空格被视为单个空格。输出 Text 类的对象时，要在 StringVar 类型的每个值之间插入一个空格。可假定其中没有使用制表符，或者干脆像对待空格那样对待制表符。如果是课堂作业，请询问老师如何对待制表符。

   添加如编程项目 4 所述的改进措施。提取操作符>>的重载版本只填充动态数组的一个元素。

6. 使用动态数组实现多项式类，它支持多项式加法、减法和乘法。

   讨论：多项式中的一个变量只是作为系数的占位符使用。因此，多项式的关键在于系数及其相应指数的一个数组。例如，以下多项式：

   $$x*x*x + x + 1$$

   要实现多项式类，一个简单的办法就是使用一个 double 数组来存储系数。数组的索引就是相应的那个项的指数。注意，上例缺少 $x*x$，它到哪里去了？如果缺少一个项，表明其系数为 0。

   可采取某些技术来表示缺少大量项的高阶多项式，它们统称为“稀疏多项式”技术。但是，除非你已经弄懂了这些技术，或者能够很快学会，否则不要使用它们。

   请提供一个默认构造函数、一个拷贝构造函数以及一个参数化构造函数，从而允许创建任意多项式。还要提供一个重载的操作符=以及一个析构函数。

   请提供以下运算：

   - 多项式 + 多项式
   - 常量 + 多项式
   - 多项式 + 常量
   - 多项式 - 多项式
   - 常量 - 多项式
   - 多项式 - 常量
   - 多项式 * 多项式
   - 常量 * 多项式
   - 多项式 * 常量

   提供函数来指派和提取系数，这些系数根据指数进行索引。

   提供一个函数对多项式进行求值，结果是 double 值。

至于这些函数是作为成员、友元还是独立的函数来实现，要由你自己决定。

7. 写支票簿结算程序。程序要读取自从上一次结算支票簿以来，没有兑付的所有支票的数据。这些数据包括：每张支票的号码、支票的金额以及它是否兑付。使用一个以类作为基类型的数组。类的每个对象都表示一张支票。应该用 3 个成员变量来记录支票号码、支票金额以及支票是否兑付。支票类使用一个 Money 类型(在图 11.9 中定义)的成员变量来记录支票金额。换言之，需要在一个类中使用另一个类。支票类要提供相应的取值和赋值函数，还要提供恰当的构造函数，以及用于输入和输出的函数。

   除了支票，程序还应读取所有储蓄金额，以及旧的和新的账户余额。可考虑用另一个数组来存储储蓄。新的账户余额等于旧账户余额加上所有储蓄金额，再减去已兑付的所有支票的金额。

   程序输出的是已兑付的所有支票的总金额、储蓄总金额、新的账户余额以及它与银行报告的余额的差值。它还要输出两个支票列表：自从上一次结算支票簿以来兑付的支票，以及仍然没有兑付的支票。显示支票列表时，根据支票号码从小到大进行排序。

   如果这是课堂作业，请询问老师输入/输出是应该用键盘和屏幕完成，还是用文件。用文件还要问清楚文件名。

8. 定义 List 类，其对象用于存储 double 类型的一组值。根据图 11.10 给出的 TemperatureList 类创建类定义。但 List 类在输出值时，不能和温度产生任何联系。它的值可表示任何种类的数据项，只要是 double 类型就可以。添加自测题 21 和自测题 22 要求的附加功能。更改成员函数名称，去掉它们和温度的联系。

   添加名为 getLast 的成员函数，该函数没有任何参数，返回列表最后一项。成员函数 getLast 不会更改列表。但是，如果列表为空，就不应该调用成员函数 getLast。再添加名为 deleteLast 的成员函数，它删除列表最后一个元素。成员函数 deleteLast 是 void 函数。注意，删除最后一个元素之后，成员变量 size 也必须相应调整。调用 deleteLast 时，如果调用对象是空列表，函数就不采取任何操作。设计一个程序全面测试 List 类的定义。

9. 定义 StringSet 类，其对象用于存储一个 STL 字符串集合。用一个数组或者向量来存储字符串。创建一个构造函数，该函数获取一个字符串数组作为参数，作用是设置集合的初始值。然后，写一组成员函数，分别用于：在集合中添加一个字符串，从集合中删除一个字符串，清除整个集合，返回集合中的字符串数量，以及输出集合中的所有字符串。重载+操作符，使它返回两个 StringSet 对象的并集。同时重载*操作符，使它返回两个 StringSet 对象的交集。写一个程序来测试你的类中的所有成员函数和重载的操作符。

10. 这个编程项目要求先完成编程项目 9。

   在信息检索领域，人们关心如何基于一个查询来找到相关的电子文档。例如，给定一组关键字(查询)，一个搜索引擎应当检索网页(文档)，根据它们与查询关键字的相关度排序并向用户显示。该技术要求采取一种方式将文档与查询关键字进行对比，判断与查询关键字最相关的是什么。
   为了执行这种比较，一个简单的方式是计算 binary cosine 系数。这个系数是 0～1 的一个值；其中 1 表示查询与文档非常相似；0 表示查询没有关键字与文档相关。这个方法将每个文档视为一个单词集合。例如，假定有下面这个示范文档：

   “chocolate ice cream, chocolate milk, and chocolate bars are delicious.”

   这个文档会被解析成一组关键字，它们的大小写会被忽略，标点符号会被丢弃，最终获得的是一个字符串集合：{chocolate, ice, cream, milk, and, bars, are, delicious}。对于用户输入的查询关键字，会按照完全相同的过程进行处理，将其转换成一个字符串集合。

   现在，假定你有一个表示成单词集合的查询 Q，以及同样表示成一个单词集合的文档 D，那么 Q 和 D 的相似性是这样计算的：

$$Sim = \frac{|Q \cap D|}{\sqrt{|Q|}\sqrt{|D|}}$$

   修改编程项目 9 的 StringSet 类，新增一个成员函数来计算当前 StringSet 与 StringSet 类型的一个输入参数的相似性。sqrt 函数在 cmath 库中。

在磁盘上创建两个文本文件，命名为 Document1.txt 和 Document2.txt。在每个文件中都添加你选择的一些文本内容，但确保每个文件包含的都是不同内容。接着写一个程序，允许用户从键盘输入一组字符串来代表一个查询。然后，程序应该将查询与磁盘上的两个文本文件进行比较，并使用以上公式来输出各自的相似性。使用不同的查询来测试程序，检查相似性报告结果是否准确。

11. 重做第 9 章的编程项目 6(或者第一次做)，但这次要求将动态数组和数组大小封装到一个类中。类应该提供 public 成员函数 addEntry(添加项)和 deleteEntry(删除项)。代表数组和数组大小的变量则应是 private 的。这就要求添加函数来做下面这些事情：获取和设置数组中特定的项，以及返回数组当前大小。添加析构函数来释放为动态数组分配的内存。另外，添加拷贝构造函数，并重载赋值操作符，使动态数组从赋值操作符右侧的对象正确复制到左侧的对象。将类嵌入合适的测试程序。

12. 视频讲解：*Solution to Programming Project 11.12*

为防止选举作弊，城市建立了一套新的选举规程。每张选票为投票人可能做出的每项选择都关联了一个字母。下面是一张示范选票：

1. VOTE FOR MAYOR
   A.　Pincher, Penny　☐
   B.　Dover, Skip　☐
   C.　Perman, Sue　☐
2. PROPOSITION 17
   D.　YES　☐
   E.　NO　☐
3. MEASURE 1
   F.　YES　☐
   G.　NO　☐
4. MEASURE 2
   H.　YES　☐
   I.　NO　☐

提交选票后，每个投票人都会获得一纸收据，上面有独一无二的 ID 号，以及反映其选择的一个记录。例如，假定一个投票人在自己的选票上投了 Sue Perman 一票，为 Proposition 17 选择 Yes，为 Measure 1 选择 No，为 Measure 2 选择 Yes，则收据可能这样写：

```
ID 4925 :   CDGH
```

第二天，市政厅会在主页上发布所有市民的投票情况，这些记录按 ID 号排序。这样投票人就能确认自己的选择，而且所有人都能根据网站上的数据自行汇总结果。下面是一个示范选票列表：

ID	VOTES
4925	CDGH
4926	AEGH
4927	CDGI
4928	BEGI
4929	ADFH

写程序从文件读取市政厅公布的选票列表，输出每个投票选项的投票百分比。可假定文件没有任何标题行。第一行就包含投票人 ID 和代表投票情况的字符串。定义 Voter 类来存储个人投票记录。类的一个构造函数能获取一个选举字符串(例如"CDGH")和一个投票人 ID 作为输入。类还应该有恰当的取值函数，能返回投票人的 ID 以及他/她对一个特定的选举项的投票情况。将每个 Voter 实例都存储到一个数组或向量中。程序应该遍历这个数组或向量，计算并输出每个候选人、每个 proposition 和 measure 的选票百分比。然后提示用户输入一个投票人 ID，再次遍历列表/向量来查找具有该 ID 的对象，并打印他/她的投票情况。

13. 重做第 10 章的编程项目 11，使用数组而不是单独的变量存储电影打分。所有更改都在类的内部进行，用于测试类的 main 函数分辨不出自己使用的是旧 Movie 类，还是包含了数组成员变量的新 Movie 类。

然后修改 main 函数，不是为每个 Movie 对象创建单独的变量，而是创建包含 4 个 Movie 对象的数组，并填充示范数据。遍历数组，输出每部电影的名称、MPAA 分级以及平均分。

14. 重做第 8 章的编程项目 16，这次用 Racer 类存储运动员信息，包括姓名、号码、最终排名以及由 RFID

传感器记录的所有分段计时。可选择合适的结构来存储这些信息。包括恰当的函数来获取或更改运动员信息，还要添加一个构造函数。创建包含 `Racer` 对象的数组或向量来存储完整比赛结果。

运动员姓名应来自一个单独的文本文件。选手注册时就收集好了这些信息。下面是一个示范文件：

100, Bill Rodgers
132, Frank Shorter
182, Joan Benoit

15. 重做第 8 章的编程项目 19，这次用 `Player` 类存储玩家名字和分数。使用 `Player` 类型的单个数组或向量。包括一个构造函数来设置名字和分数。

第 12 章　独立编译和命名空间

(他又知道我爱好书籍，)特意从我的书斋里把那些我看得比一个公国更宝贵的书给我带了来。[①]

——*威廉·莎士比亚，《暴风雨》第 1 幕第 2 场*

概述

本章的两个主题解释了如何将 C++程序组织成独立的部分。12.1 节介绍独立编译，讨论如何将 C++程序分布到多个文件中。这样一来，一旦程序的某些部分发生改变，只需重新编译那些部分。独立的部分在其他应用程序中也可以方便地重用。

12.2 节介绍命名空间，第 2 章已简单讨论了这个主题。利用命名空间，可以重用类、函数和其他项目的名称。只需对这些名称进行限定，指明不同的用途。命名空间将代码分解为不同小节，使不同小节能重用同一个名称，并各自为该名称赋予不同含义。命名空间为那些比局部变量更一般化的名称(比如 cin)赋予了某种局部的含义。

预备知识

本章基于第 2 章～第 6 章和第 10 章～第 11 章的知识。

12.1 独 立 编 译

“假如”是唯一的和事佬；“假如”之用为大矣哉！[②]

——*威廉·莎士比亚，《皆大欢喜》第 5 幕第 4 场*

C++允许将程序分解成多个部分，保存到独立的文件中，独立编译，并在程序运行时(或在运行前)链接到一起。可将类定义(以及相关函数定义)放到一个文件中，使这个文件独立于使用该类的程序。这样就可方便地建立起一个类库，以便多个程序使用相同的类。类只需编译一次，即可在多个程序中使用，和使用预定义库(比如头文件是 iostream 和 cstdlib 的那些库)一样。此外，类也可用两个文件来定义，将这个类的使用规范与类的实现细节区分开。如果类按规则定义，以后只需重新编译类的实现文件。其他文件(包括使用类的程序文件)不需要更改，甚至不需要重新编译。本节将解释具体如何实现类的独立编译。

ADT 回顾

10.3 节讲过，ADT(抽象数据类型)将类的接口和实现区分开。所有类定义都应该是 ADT。为了定义 ADT 类，需要将类的使用规范(这是给类的使用者，也就是其他程序员看的)与类的实现细节分开。这种分离应该非常彻底，目的是更改了实现之后，完全不用更改正在使用类的任何程序。遵循以下三个规则可确保彻底分离。

其一，使所有成员变量都成为类的私有成员。

---

① 书斋和库在英语中都是 library。——译注

② “假如”原文是 if，跟 12.1 节要讲的#ifndef 条件编译指令有莫大关系。——译注

其二，使 ADT 类的每一项基本操作都成为类的公共成员函数、友元函数、普通函数或者重载操作符。将类定义与函数/操作符声明合并。这个组别(以及所有相关注释)统称为 ADT 的**接口**。在类或函数/操作符的注释中，完整说明如何使用每一个函数或操作符。

其三，确保使用 ADT 的程序员访问不到基本操作的具体实现。**实现**包括函数定义以及重载操作符的定义(另外包括任何辅助函数，或者这些定义需要的其他项)。

在 C++中，为了确保遵循这些规则，最好的办法就是将 ADT 的接口与实现存储到独立的文件中。你可能已经猜到，包含接口的文件称为**接口文件**(interface file)，包含实现的文件称为**实现文件**(implementation file)。不同版本的 C++可能要以不同方式来设置、编译和使用这些文件。但基本方案在所有版本的 C++中都一样。特别注意，在所有系统中，文件具体存储的内容都一样。唯一区别是用什么命令来编译和链接这些文件。下一节的案例分析将具体展示这些文件所存储的内容。

ADT 类含有私有成员变量。私有成员变量(和私有成员函数)为 ADT 的接口与实现分离原则带来了一个问题。ADT 类定义的 `public` 部分是 ADT 接口的一部分，但 `private` 部分是实现的一部分。之所以成为问题，是由于 C++不允许将类定义拆分成两个文件。所以必须有一个折衷方案。唯一可行 (也是我们准备采用) 的折衷方案是将整个类定义都放在接口文件中。由于使用 ADT 类的程序员不能使用类的任何私有成员，所以在程序员面前，私有成员实际上还是隐藏的。

> **ADT**
>
> 如果使用数据类型的程序员访问不到值和操作的实现细节，就可将这种数据类型称为抽象数据类型(ADT)。你定义的所有类都应该是 ADT。ADT 类确保了类的接口和实现的分离，这是良好的编程实践(类的任何非成员基本操作——比如重载的操作符——也被视为 ADT 的一部分，即使它们可能并非类定义的正式部分)。

案例分析：独立编译的 DigitalTime 类

图 12.1 展示了 ADT 类 `DigitalTime` 的接口文件。`DigitalTime` 类表示一天中的时间，比如 9:30。只有类的公共成员才是接口的一部分。私有成员是实现的一部分，即使它们也在接口文件中列出。标签 `private:`提醒你这些私有成员不是公共接口的一部分。程序员使用 ADT 类 `DigitalTime` 时，需要知道的一切都在文件开头的注释以及类定义的 `public` 小节的注释中进行解释。接口告诉程序员如何使用成员函数 `advance` 的两个版本、各个构造函数以及重载的操作符==、>>和<<。程序员操纵这个类的对象和值时，只能使用名为 `advance` 的成员函数、重载的操作符以及赋值语句。正如接口文件开头的注释所指明的，这个 ADT 类使用 24 小时制。例如，假定输入 `1:30 PM`，输出就是 `13:30`。在成员函数的注释中，包含有效使用 `DigitalTime` 类需了解的一切细节。

图 12.1　DigitalTime 类的接口文件

```
1  // 头文件 dtime.h：这是 DigitalTime 类的“接口”部分
2  // 这个类型的值表示一天当中的时间。值采取 24 小时制进行输入和输出
3  // 例如，9:30 表示 9:30 AM，14:45 表示 2:45 PM
```

```
 4 #include <iostream>          ←—— 这是为了使用类型 istream 和 ostream
 5 using namespace std;              的定义。这两个类型要用作参数类型

 6 class DigitalTime
 7 {
 8 public:
 9     friend bool operator ==(const DigitalTime& time1, const DigitalTime& time2);
10     // 如果 time1 和 time2 代表相同的时间，就返回 true；否则返回 false

11     DigitalTime(int theHour, int theMinute);
12     // 前条件：0 <= theHour <= 23，而且 0 <= theMinute <= 59
13     // 将时间值初始化为 theHour 和 theMinute

14     DigitalTime();
15     // 将时间值初始化为 0:00(午夜)

16     void advance(int minutesAdded);
17     // 前条件：对象已经有一个时间值
18     // 后条件：时间被更改为 minutesAdded 分钟之后

19     void advance(int hoursAdded, int minutesAdded);
20     // 前条件：对象已经有一个时间值
21     // 后条件：将时间值调后 hoursAdded 小时加 minutesAdded 分钟

22     friend istream& operator >>(istream& ins, DigitalTime& theObject);
23     // 重载>>操作符，允许输入 DigitalTime 类型的值
24     // 前条件：如果 ins 是一个文件输入流，那么 ins 已经与一个文件连接

25     friend ostream& operator <<(ostream& outs, const DigitalTime& theObject);
26     // 重载<<操作符，允许输出 DigitalTime 类型的值
27     // 前条件：如果 outs 是一个文件输出流，那么 outs 已经与一个文件连接
28 private:  }
29     int hour;  } ←—— 这是实现的一部分，不是接口的一部分。修饰符
30     int minute; }    private 表明这不是公共接口的一部分
31 };
```

我们将接口放到名为 dtime.h 的文件中。扩展名.h 表明这是一个头文件。所有接口文件都是头文件，所以文件名肯定有扩展名.h。要使用 DigitalTime 类的所有程序都必须包含下面这个 include 预编译指令，它指定了该文件的名称：

```
#include "dtime.h"
```

写一个 include 预编译指令时，必须指明头文件是预定义头文件，还是你自己编写的头文件。如果头文件是预定义的，就将头文件名放到一对尖括号中，比如<iostream>。如果头文件是自己写的，就将头文件名放到一对引号中，比如"dtime.h"。这样一来，编译器就知道应该在哪里寻找头文件。如果头文件名在尖括号中，编译器会在存放预定义头文件的地方(这取决于具体的 C++实现)寻找指定头文件。如果头文件名在引号中，编译器会在当前目录或者存放“程序员自定义头文件”的其他地方寻找指定头文件。

任何使用了这个 DigitalTime 类的程序都必须包含前面那个命名了头文件 dtime.h 的 include 预编译指令。现在能编译程序，但还不能运行程序。要运行程序，还必须编写(及编译)成员函数以及重载的操作符的定义。我们将这些函数和操作符定义放在另一个文件中，该文件称为**实现文件**。虽然大多数编译器都不做要求，但根据传统，接口文件和实现文件应该同名。当然，必须用不同的扩展名。我们已将当前这个 ADT 类的接口放在名为 dtime.h 的文件中，它的实现则准备放在名为 dtime.cpp 的文件中。具体为实现文件采用什么扩展名，要取决于你所用的 C++版本。通常，为包含 C++程序的文件用什么扩展名，就

为实现文件用什么扩展名。如果程序文件名以.cxx 结尾，就用.cxx 代替.cpp。如果程序文件名以.CPP 结尾，实现文件也要采用.CPP 作为扩展名，而不是.cpp。我们使用.cpp，因为大多数编译器都将.cpp 视为 C++源代码文件扩展名。DigitalTime ADT 类的实现文件在图 12.2 中给出。解释了 ADT 的各个文件相互之间如何交互之后，我们会回到图 12.2，讨论实现文件中的定义细节。

图 12.2　DigitalTime 的实现文件

```
 1  // 实现文件 dtime.cpp(你的系统可能要求不用于.cpp 的一个扩展名):
 2  // 这是 ADT 类 DigitalTime 的“实现”部分
 3  // DigitalTime 类的接口放在头文件 dtime.h 中
 4  #include <iostream>
 5  #include <cctype>
 6  #include <cstdlib>
 7  #include "dtime.h"
 8  using namespace std;

 9  // 以下函数声明在重载的输入操作符>>的定义中使用:

10  void readHour(istream& ins, int& theHour);
11  // 前条件: 流 ins 中的下一个输入是一个采用 24 小时制的时间, 比如 9:45 或 14:45
12  // 后条件: theHour 被设为时间的小时部分. 冒号被舍弃, 读取的下一个输入将是分钟数

13  void readMinute(istream& ins, int& theMinute);
14  // readHour 函数读取了小时数之后, 这个函数从流 ins 中读取分钟数

15  int digitToInt(char c);
16  // 前条件: c 是'0'～'9'的一个数字
17  // 返回数字的整数值; 例如, digitToInt('3')返回 3

18  bool operator ==(const DigitalTime& time1, const DigitalTime& time2)
19  {
20      return (time1.hour == time2.hour && time1.minute == time2.minute);
21  }

22  // 使用 iostream 和 cstdlib:
23  DigitalTime::DigitalTime(int theHour, int theMinute)
24  {
25      if (theHour < 0 || theHour > 23 || theMinute < 0 || theMinute > 59)
26      {
27          cout << "Illegal argument to DigitalTime constructor.";
28          exit(1);
29      }
30      else
31      {
32          hour = theHour;
33          minute = theMinute;
34      }
35  }

36  DigitalTime::DigitalTime() : hour(0), minute(0)
37  {
38      // 主体有意留空
39  }

40  void DigitalTime::advance(int minutesAdded)
41  {
42      int grossMinutes = minute + minutesAdded;
43      minute = grossMinutes%60;

44      int hourAdjustment = grossMinutes/60;
45      hour = (hour + hourAdjustment)%24;
46  }
```

```
47 void DigitalTime::advance(int hoursAdded, int minutesAdded)
48 {
49     hour = (hour + hoursAdded) % 24;
50     advance(minutesAdded);
51 }
52 // 使用 iostream:
53 ostream& operator <<(ostream& outs, const DigitalTime& theObject)
54 {
55     outs << theObject.hour << ':';
56     if (theObject.minute < 10)
57         outs << '0';
58     outs << theObject.minute;
59     return outs;
60 }
61 // 使用 iostream:
62 istream& operator >>(istream& ins, DigitalTime& theObject)
63 {
64     readHour(ins, theObject.hour);
65     readMinute(ins, theObject.minute);
66     return ins;
67 }
68 int digitToInt(char c)
69 {
70     return (staticCast<int>(c) - staticCast<int>('0'));
71 }
72 // 使用 iostream, cctype 和 cstdlib:
73 void readMinute(istream& ins, int& theMinute)
74 {
75     char c1, c2;
76     ins >> c1 >> c2;
77     if (!(isdigit(c1) && isdigit(c2)))
78     {
79         cout << "Error illegal input to readMinute\n";
80         exit(1);
81     }
82     theMinute = (digitToInt(c1) * 10) + digitToInt(c2);
83     if (theMinute < 0 || theMinute > 59)
84     {
85         cout << "Error illegal input to readMinute\n";
86         exit(1);
87     }
88 }
89 // 使用 iostream, cctype 和 cstdlib:
90 void readHour(istream& ins, int& theHour)
91 {
92     char c1, c2;
93     ins >> c1 >> c2;
94     if ( !( isdigit(c1) && (isdigit(c2) || c2 == ':' ) ) )
95     {
96         cout << "Error illegal input to readHour\n";
97         exit(1);
98     }
99     if (isdigit(c1) && c2 == ':')
100    {
101        theHour = digitToInt(c1);
102    }
103    else //(isdigit(c1) && isdigit(c2))
104    {
105        theHour = (digitToInt(c1) * 10) + digitToInt(c2);
```

```
106         ins >> c2; // 舍弃':'
107         if (c2 != ':')
108         {
109            cout << "Error illegal input to readHour\n";
110            exit(1);
111         }
112      }
113      if ( theHour < 0 || theHour > 23 )
114      {
115         cout << "Error illegal input to readHour\n";
116         exit(1);
117      }
118  }
```

程序要使用 ADT 类 DigitalTime，必须包含 include 预编译指令：

```
#include "dtime.h"
```

注意，无论实现文件还是程序文件，都必须包含上述命名了接口文件的 include 预编译指令。包含程序的文件(也就是包含程序 main 部分的文件)通常称为**应用程序文件**或**驱动程序文件**。图 12.3 展示了一个应用程序文件，这是很简单的程序，它使用并演示了 ADT 类 DigitalTime。

图 12.3　使用 DigitalTime 类的应用程序文件

```
1  // 应用程序文件 timedemo.cpp(你的系统可能要求和.cpp 不同的一个扩展名):
2  // 该程序演示了 DigitalTime 类的使用
3  #include <iostream>
4  #include "dtime.h"
5  using namespace std;
6
7  int main()
8  {
9     DigitalTime clock, oldClock;
10
11    cout << "Enter the time in 24 hour notation: ";
12    cin >> clock;
13
14    oldClock = clock;
15    clock.advance(15);
16    if (clock == oldClock)
17       cout << "Something is wrong.";
18    cout << "You entered " << oldClock << endl;
19    cout << "15 minutes later the time will be "
20         << clock << endl;
21
22    clock.advance(2, 15);
23    cout << "2 hours and 15 minutes after that\n"
24         << "the time will be "
25         << clock << endl;
26
27    return 0;
28 }
```

示范对话

```
Enter the time in 24-hour notation: 11:15
You entered 11:15
15 minutes later the time will be 11:30
2 hours and 15 minutes after that
the time will be 13:45
```

具体怎样运行这个由三个文件构成的完整程序，取决于所用的系统。但所有系统的基本原理都一样。首先编译实现文件，然后编译包含程序 main 部分的应用程序文件。不需要编译接口文件(图 12.1 展示的 dtime.h 文件)，因为编译器认为这个接口文件的内容已包含在其他两个文件中。记住，实现文件和应用程序文件都包含以下预编译指令：

```
#include "dtime.h"
```

编译程序时，会自动调用一个预处理器。它读取这个 include 指令，把它替换成文件 dtime.h 包含的文本。因此，编译器最终看见的是 dtime.h 的内容，所以 dtime.h 文件不需要独立编译(事实上，编译器会分两次看到 dtime.h 的内容：一次在编译实现文件时，一次在编译应用程序文件时)。对 dtime.h 文件内容的复制只是概念上的复制。编译器表现得将 dtime.h 的内容复制到含有 include 预编译指令的每个文件中。但实际上，编译之后再去看那个文件，看见的仍然只是 include 预编译指令，不会看见文件 dtime.h 的内容。

编译好实现文件和应用程序文件后，还要连接这些文件，使它们能共同工作。这称为对文件进行**链接**(linking)，由一个独立的实用程序(名为**链接器**，即 linker)完成。至于具体怎样调用链接器，要取决于所用的系统。文件链接好之后，就可运行最终生成的可执行程序(通常，链接作为程序编译过程的一部分来完成)。

这个过程表面上很复杂，但许多系统都提供了相应的机制来帮助你管理这些细节，能自动或半自动地完成大多数操作。对于任何系统，只要有意去熟悉它的工作方式，就能迅速掌握它的所有细节。

图 12.1、图 12.2 和图 12.3 包含一个完整的程序，该程序被分解成不同部分，并用 3 个独立的文件来存储。也可将这 3 个文件的内容合并成一个文件，并编译和运行该文件，而不是麻烦地使用这些 include 预编译指令，并对独立的文件进行链接。那么，使用 3 个独立的文件到底有什么好处？将程序分解成独立的文件具有几方面的好处。由于 DigitalTime 类的定义和实现分别放在独立的文件中，并与应用程序文件区分开，所以能在许多不同的程序中使用这个类，不需要在每个程序中重写类的完整定义。除此之外，无论有多少个程序使用 DigitalTime 类，实现文件只需编译一次。除此之外，还有另一些好处。由于将 ADT 类 DigitalTime 的接口和实现分开，所以能更改实现文件，同时不需要更改任何使用 ADT 的程序。事实上，甚至不需要重新编译程序。更改了实现文件，只需重新编译实现文件，并重新链接文件。除了节省一定的重新编译时间，最大的好处是不必重写代码。可在许多程序中直接使用 ADT 类，不需要将类的代码插入每个程序。可以更改 ADT 类的实现，不需要重写使用了该类的任何程序的任何部分。

前面解释了 ADT 类中的各个文件以及程序如何使用。接着具体讨论图 12.2 的 ADT 类的实现。大多数实现细节直观易懂，但其中两个要点需要特别留意。首先，成员函数名 advance 进行了重载，它拥有了两个函数定义。其次，在重载的提取(输入)操作符>>的定义中，我们使用了两个辅助函数，分别是 readHour 和 readMinute。这两个辅助函数本身又使用了第 3 个辅助函数，名为 digitToInt。下面具体讨论这些要点。

总结：用多个独立的文件定义类

可定义一个类，将类的定义及其成员函数的实现放到各自独立的文件中。然后独立于任何使用该类的程序来编译这个类，并可在任意数量的程序中使用这个类。类和使用该类的程序放到 3 个独立的文件中，如下所示。

- 类的定义放到一个头文件中，这个头文件称为**接口文件**。头文件名以.h 结尾。任何函数和重载的操作符，如果它们定义了类的基本操作，但不在类定义中列出，它们的声明也要放到接口文件中。需添加相应的注释，解释所有这些函数和操作符应该如何使用。
- 前面提到的所有函数以及重载的操作符的定义(无论它们是成员还是友元，或者两者都不是)都要放到另一个文件中，该文件称为**实现文件**。该文件必须包含一个 include 预编译指令，并指定上述接口文件的名称。这个 include 预编译指令用引号封闭文件名，如下例所示：

```
#include "dtime.h"
```

  根据传统，接口文件和实现文件应该同名，但以不同的扩展名结尾。接口文件的扩展名是.h，实现文件的扩展名与包含普通 C++程序的扩展名相同。要先单独编译实现文件，然后才能在其他程序中使用它。
- 如果希望在程序中使用类，需要将程序 main 部分(以及任何附加的函数定义、常量声明等)放到另一个文件中，这个文件称为**应用程序文件**。文件中也必须包含一个命名了接口文件的 include 指令，如下例所示：

```
#include "dtime.h"
```

应用程序文件独立于实现文件进行编译。可以写任意数量的应用程序文件，并使用这一对接口文件和实现文件。要运行完整的程序，首先必须将编译应用程序文件生成的目标码与编译实现文件生成的目标码链接起来(某些系统上，这个链接过程自动或半自动完成)。

.h 和.hpp

有的 C++程序员和 IDE 选择为接口文件使用.hpp 扩展名而不是.h。.hpp 的一个好处是一眼就能看出这是 C++文件。.h 最初为 C 所用。如某个项目兼具 C 和 C++代码，则.hpp 有利于区分。有的程序员还将实现放到.hpp 文件中。由模板和内联函数构成的库更加如此。(模板在第 17 章讨论。)

DigitalTime 类(图 12.1 和图 12.2)有两个名为 advance 的成员函数。一个版本获取单个参数，这是一个整数值，指定要使时间向后调的分钟数。另一个版本获取两个参数，一个代表小时数，另一个代表分钟数，它们使时间向后调那个小时数加上那个分钟数。注意，在 advance 的双参数版本的定义中，包括了对 advance 的单参数版本的调用。注意图 12.2 中的双参数版本的定义。首先，时间调后 hoursAdded 小时，然后调用 advance 的单参数

版本，使时间再调后额外的 minutesAdded 分钟。表面看有些奇怪，但这完全合法。两个函数的名称虽然都是 advance，但从编译器的角度看，它们是两个不同的函数，两者只是凑巧名称相同。如果不是进行重载，而是将一个命名为 advance，另一个命名为 anotherAdvance，情况也不会有任何分别。

下面讨论辅助函数。辅助函数 readHour 和 readMinute 每次从输入中读取一个字符，将输入转换成相应整数值，再将值存储到成员函数 hour 和 minute 中。readHour 和 readMinute 每次读取小时和分钟数的一个数位，所以它们读取的是 char 类型的值。相较于直接以 int 值的形式读取输入，这当然会复杂一些，但便于我们执行错误检查，了解输入格式是否正确，并在输入不正确的前提下报错。辅助函数 readHour 和 readMinute 使用了另一个辅助函数 digitToInt，它与图 11.3 的 Money 类定义中使用的 digitToInt 函数一样。digitToInt 函数的作用是将一个 char 类型的数位(比如'3')转换成 int 类型的数值(比如 3)。

可重用组件

开发成功并用独立的文件来存储的 ADT 类是一个有用的软件组件，可在许多不同的程序中重复使用。设计软件组件时，非常重要的一个目标便是获得这种**重用性**。

可重用组件能极大节省时间和精力，因为以后不需要为每个应用程序都重新设计、记录和测试它。基于两方面的原因，可重用组件的可靠性也要优于一次性组件。首先，由于知道一个组件会多次使用，所以会花较多时间和精力去完善。其次，由于组件会重复使用，所以会一次又一次地得到测试。

每次使用软件组件时，都相当于对该组件进行一次测试。为了查找软件中遗留的任何 bug，在多种场合中重复使用同一个软件组件是很好的除错方法。

使用#ifndef

视频讲解：*Avoiding Multiple Definitions*

前面解释了用 3 个文件容纳一个程序的方法。其中，两个文件分别存储类的接口和实现部分，另一个存储应用程序部分。但事实上，一个程序可以使用不止三个文件容纳。例如，程序可能使用了几个类，每个类都用两个独立的文件容纳。假定程序分布在多个文件中，而且不止一个文件包含一个类的接口文件，例如：

```
#include "dtime.h"
```

在这些情况下，一些文件可能包含其他文件，而其他文件又包含更多的文件。结果就是，一个文件最终可能多次包含 dtime.h 的定义。C++不允许多次定义同一个类，即使重复的是完全相同的定义。此外，在多个项目中使用同一个头文件，几乎不可能知道一个类的定义是否被多次包含。为避免这个问题，C++提供了一种方式标记一个代码小节，指出：“如果以前已包含了这一节的内容，就不要再次包含。”具体操作是非常直观的。但除非习惯，否则刚开始会觉得所用的记号法有一点儿“怪”。下面用一个例子解释需注意的细节。

以下预编译指令“定义”DTIME_H：

```
#define DTIME_H
```

它要求计算机的预处理器将 DTIME_H 放到一个列表中，指出已经看见了 DTIME_H。“定义”也许不是最理想的词，因为没有定义 DTIME_H 来表示任何东西——只是把它放到列表中。重点在于，可用另一个预编译指令测试 DTIME_H 是否已“定义”，判断一个代码小节是否已得到处理。虽然可用任何(非关键字)标识符取代 DTIME_H，但稍后会讲到，标识符的使用存在一些标准规范，你应该遵守这些规范。

以下预编译指令用于测试 DTIME_H 是否已定义：

```
#ifndef DTIME_H
```

如 DTIME_H 已定义，以上预编译指令和以下预编译指令之间的所有内容都会被忽略：

```
#endif
```

为了澄清预编译指令的拼写方式，可以换一种说法：如果 DTIME_H 没有定义，编译器就要处理直到下一个#endif 的所有内容。注意，ifndef 的意思是 if not def，也就是“如果没有定义”。你自然会问是否有对应的#ifdef 预编译指令。确实有，而且它的意思不言而喻，只是暂时没有机会用到#ifdef。

现在分析以下代码：

```
#ifndef DTIME_H
#define DTIME_H
<一个类定义>
#endif
```

如果上述代码在一个名为 dtime.h 的文件中，那么程序无论包含了以下语句多少次，类都只定义一次：

```
#include "dtime.h"
```

第一次处理上述语句，会正常定义 DTIME_H 标志(#define DTIME_H)，类也会正常定义。以后，如果编译器再次遇到：

```
#include "dtime.h"
```

由于 include 预编译指令是第二次进行处理，所以以下预编译指令：

```
#ifndef DTIME_H
```

会忽略直到以下语句之前的一切内容：

```
#endif
```

所以类不会重复定义。

图 12.4 改写了头文件图 12.1 的头文件 dtime.h。

图 12.4　避免多次定义一个类

```
1  // 头文件 dtime.h：这是 DigitalTime 类的“接口”部分
2  // 这个类型的值表示一天当中的时间。值采取 24 小时制进行输入和输出
3  // 例如，9:30 表示 9:30 AM，14:45 表示 2:45 PM
4  #ifndef DTIME_H
5  #define DTIME_H
6  #include <iostream>
7  using namespace std;
8  class DigitalTime
```

```
 9 {
10 <DigitalTime 类的定义与图 12.1 相同>
11 };
12
13 #endif // DTIME_H
```

这一次使用预编译指令防止重复定义。使用图 12.4 中的 dtime.h，如果文件中多次包含以下 include 预编译指令，DigitalTime 类只会定义一次：

```
#include "dtime.h"
```

可用其他标识符代替 DTIME_H，但根据规范，标识符应以文件名为准，全部换成大写字母，同时用下划线取代句点符号。应遵循这一规范，使其他人更容易阅读你的代码，你自己也不必记忆标志名称。这样一来，标志名称能自动确定，不必记忆多余的东西。

利用同样的预编译指令，还可在除了头文件之外的其他文件中跳过特定代码。但是，本书只会在头文件中使用这些预编译指令。

> **#pragma once**
>
> 虽然不是标准，但大多数 C++编译器都支持预编译指令#pragma once。添加到文件顶部，源代码文件编译时就只包含一次。有时用它代替#ifndef 构造。

编程提示：定义其他库

不定义类也能使用独立编译技术。要将一套相关的函数放到单独的库中，可将函数声明及其注释放到头文件，将函数定义放到实现文件，具体操作和之前为 ADT 类描述的操作相同。之后就可像使用放在独立文件中的类那样，在自己的程序中使用该函数库。■

√ 自测题

1. 假定要定义一个 ADT 类，并在程序中使用该类。你希望参照本章的描述，用独立的文件存储类和程序。给定以下项目，请说明它们是应该放到接口文件中，还是应该放到实现文件中，或者应该放到应用程序文件中？
   a. 类定义。
   b. 函数声明，该函数是 ADT 的一个操作，但它不是类的成员，也不是类的友元。
   c. 重载操作符声明，这个重载的操作符是 ADT 的一个操作，但它不是类的成员，也不是类的友元。
   d. 函数定义，该函数是 ADT 的一个操作，但它不是类的成员，也不是类的友元。
   e. 友元函数定义，该函数是 ADT 的一个操作。
   f. 成员函数定义。
   g. 重载操作符定义，这个重载的操作符是 ADT 的一个操作，但它不是类的成员，也不是类的友元。
   h. 重载操作符定义，这个重载的操作符是 ADT 的一个操作，同时是类的友元。
   i. 程序的 main 部分。
2. 以下哪些文件的名称要以.h 结尾(采用.h 扩展名)：类的接口文件、类的实现文件或者使用类的应用程序文件？
3. 用独立文件定义类时，有一个接口文件和一个实现文件。其中什么文件需要编译？(都要编译？都不编译？只编译一个？如果只编译一个，具体是哪个？)
4. 用独立的文件定义类，并在程序使用该类。现在，假定更改了类的实现文件。以下什么文件需要重新编译：接口文件、实现文件或者应用程序文件？
5. 假定要更改图 12.1 和图 12.2 给出的 DigitalTime 类的实现。具体地说，要更改时间的记录方式。你

希望不使用两个私有变量 hour 和 minute，而是使用单个(私有)int 变量，该变量的名称是 minutes。在这个新的实现中，私有变量 minutes 将时间记录为自 0:00(午夜)以来的分钟数。换言之，1:30 要记录成 90 分钟，因为它距离午夜 90 分钟。请说明如何更改如图 12.1 和图 12.2 所示的接口和实现文件。不需要写完整文件，只需指出需修改哪些项目，并大致说明如何修改。

6. 用 C++定义的 ADT 和类有什么区别？

12.2 命 名 空 间

> 姓名本来是没有意义的；我们称之为“玫瑰”的这一种花，要是换个其他名字，它的香味还是同样的芬芳。
>
> ——*威廉·莎士比亚，《罗密欧与朱丽叶》第 2 幕第 2 场*

一个程序使用由不同程序员写的不同的类和函数时，可能出现两个程序员为两样不同的东西使用同一名称的情况。命名空间就是为了解决这一问题而设计的。命名空间是名称定义(比如类定义和变量声明)的集合。

命名空间和 using 预编译指令

前面已用过名为 std 的命名空间，其中含有标准库文件(比如 iostream 和 cstdlib)定义的所有名称。例如，将以下语句放到文件起始位置：

```
#include <iostream>
```

就会将所有名称定义(比如 cin 和 cout 等名称的定义)放到 std 命名空间。除非指定要使用 std 命名空间，否则程序不知道 std 命名空间中的任何名称。到目前为止，我们只知道用以下 using 预编译指令指定使用 std 命名空间：

```
using namespace std;
```

为了理解为什么要用这种 using 指令，最好的办法是想想不用会发生什么。不指定使用 std 命名空间，可以自己定义 cin 和 cout，让它们的含义有别于标准含义(例如，你可能希望 cin 和 cout 的行为和标准版本稍有区别，所以可能想重新定义)。它们的标准含义存储在 std 命名空间中。没有 using 预编译指令(或其他相似指令)，你的代码就完全不了解 std 命名空间，所以对你的代码来说，cin 和 cout 唯一的定义就是你为它们赋予的定义。

事实上，你写的每一句代码都在某个命名空间中。不明确指定命名空间，代码就默认在**全局命名空间**(global namespace)中。到目前为止，我们并没有明确将代码放在任何命名空间中。因此，所有代码都在全局命名空间。全局命名空间不需要 using 指令，因为它是默认命名空间，你总是在使用这个命名空间。可以这样理解：默认情况下，有一个隐式的自动化的 using 预编译指令指定你要使用全局命名空间。

注意，可同时使用多个命名空间。例如，我们总是在使用全局命名空间，同时经常要用 std 命名空间。如果一个名称在两个命名空间都进行了定义，而你同时使用了这两个命名空间，那么会发生什么？这将导致一个错误(一个编译错误或者一个运行时错误，视具体情况而定)。虽然能在两个不同的命名空间定义相同的名称，但在这种情况下，每次只能使

用其中一个命名空间。[①]但并不是与不能在同一个程序中使用两个命名空间。在同一个程序中，可以在不同的时候使用不同的命名空间。

例如，假定 ns1 和 ns2 是两个不同的命名空间，并假定 myFunction 是无参 void 函数，它在两个命名空间中以不同的方式进行了定义。以下代码是合法的：

```
{
    using namespace ns1;
    myFunction();
}
{
    using namespace ns2;
    myFunction();
}
```

第一个调用使用命名空间 ns1 给出的 myFunction 定义，第二个调用则使用命名空间 ns2 给出的 myFunction 定义。

以前说过，代码块是一组语句、声明以及其他代码，它们用花括号封闭。块起始位置的 using 预编译指令只应用于那个块。因此，第一个 using 预编译指令只应用于第一个块，第二个只应用于第二个块。一般这样说：ns1 命名空间的作用域是第一个块，ns2 命名空间的作用域是第二个块。正是因为存在该作用域规则，所以能在同一个程序中(比如在包含上述两个块的一个程序中)使用两个冲突的命名空间。

在块中使用 using 指令，这个块通常是函数体。将 using 指令放在文件起始位置(就像我们过去常做的那样)，它会应用于整个文件。总之，using 指令一般放在文件起始位置，或者放在块的起始位置。

using 预编译指令的作用域规则

using 指令的作用域是它所在的那个块(更准确地说，从 using 指令开始，一直到当前块结束)。如 using 指令在所有块的外部，它的作用域就从 using 指令开始，一直到整个文件结束。

创建命名空间

要将代码放到命名空间中，只需将它放到以下形式的命名空间分组中：

```
namespace NameSpaceName
{
    SomeCode
}
```

在代码中包含一个命名空间分组，相当于将 *SomeCode* 中定义的名称放到命名空间 *NameSpaceName* 中。要使用这些名称(实际是这些名称的定义)，就用以下 using 指令：

```
using namespace NameSpaceName;
```

例如，以下代码(摘自图 12.5)将一个函数声明放到命名空间 savitch1 中：

```
namespace savitch1
```

① 本章稍后会指出，有办法同时使用两个命名空间，即使它们包含相同的名称，但目前暂且不关心这个问题。

```
{
    void greeting();
}
```

分析图 12.5，发现 greeting 函数的定义也放到命名空间 savitch1 中。这通过以下附加的命名空间分组来实现：

```
namespace savitch1
{
    void greeting()
    {
        cout << "Hello from namespace savitch1.\n";
    }
}
```

也就是说，可以为一个命名空间使用任意数量的命名空间分组。图 12.5 为命名空间 savitch1 使用了两个命名空间分组，为命名空间 savitch2 使用了另外两个。

命名空间中定义的每个名称都可在命名空间分组内部使用。但是，名称也可由命名空间外部的代码使用。例如，命名空间 savitch1 中的函数声明和函数定义可通过 using 预编译指令来使用：

```
using namespace savitch1
```

图 12.5 对此进行了演示。

图 12.5　演示命名空间

```
1  #include <iostream>
2  using namespace std;
3
4  namespace savitch1
5  {
6      void greeting();
7  }
8
9  namespace savitch2
10 {
11     void greeting();
12 }
13
14 void bigGreeting();
15
16 int main()
17 {
18     {
19         using namespace savitch2;
20         greeting();
21     }
22
23     {
24         using namespace savitch1;
25         greeting();
26     }
27
28     bigGreeting();
29
30    return 0;
31 }
32
33 namespace savitch1
34 {
35     void greeting()
36     {
37         cout << "Hello from namespace savitch1.\n";
38     }
```

（第 19 行注释）这个代码块中的名称使用命名空间 savitch2、std 以及全局命名空间中的定义

（第 24 行注释）这个代码块中的名称使用命名空间 savitch1、std 以及全局命名空间中的定义

（第 28 行注释）这里的名称只使用 std 和全局命名空间中的定义

```
39 }
40
41 namespace savitch2
42 {
43     void greeting()
44     {
45         cout << "Greetings from namespace savitch2.\n";
46     }
47 }
48
49 void bigGreeting()
50 {
51     cout << "A Big Global Hello!\n";
52 }
```

示范对话

```
Greetings from namespace savitch2.
Hello from namespace savitch1.
A Big Global Hello!
```

自测题

7. 对于图 12.5 的程序，可以用名称 greeting 来取代 bigGreeting 吗？
8. 通过自测题 7，我们知道不能在全局命名空间中为以下函数添加定义：

   ```
   void greeting();
   ```

   但以下函数的定义可以添加到全局命名空间中吗？

   ```
   void greeting(int howMany);
   ```

9. 一个命名空间可以有多个命名空间分组吗？

限定名称

假定两个命名空间分别是 ns1 和 ns2，你想使用 ns1 中定义的函数 fun1 以及 ns2 中定义的函数 fun2。但问题在于，在 ns1 和 ns2 这两个命名空间中，都定义了名为 myFunction 的函数(假定所有函数都不获取任何参数，所以可忽略重载的情况)。在这种情况下，以下语句是不合适的，因为它们会提供有冲突的 myFunction 定义：

```
using namespace ns1;
using namespace ns2;
```

所以，需采取一种方式，指出要使用命名空间 ns1 中的 fun1 以及命名空间 ns2 中的 fun2，但不准备使用命名空间 ns1 和 ns2 中的其他任何东西。以下语句称为 using **声明**，它们能解决这个问题：

```
using ns1::fun1;
using ns2::fun2;
```

以下形式的 using 声明，会使命名空间 *NameSpace* 中的名称 *OneName*(实际是它的定义)进入可用状态，但 *NameSpace* 中的其他任何名称仍然不可用。

```
using NameSpace::OneName
```

注意，以前已经见过作用域解析操作符::。例如，图 12.2 使用了以下函数定义：

```
void DigitalTime::advance(int hoursAdded, int minutesAdded)
```

```
{
    hour = (hour + hoursAdded) % 24;
        advance(minutesAdded);
}
```

这种情况下，操作符::表示函数 advance 是为 DigitalTime 类定义的，和其他任何类中的同名函数没有任何关系。类似地，以下语句：

```
using ns1::fun1;
```

表示要使用命名空间 ns1 定义的名为 fun1 的那个函数，而不是使用其他任何命名空间中的 fun1 定义。

现在假定要使用 ns1 中定义的 fun1，但只用一次(或用少数几次)。这种情况下，可在函数名前直接附加命名空间的名称以及作用域解析操作符，如下所示：

```
ns1::fun1();
```

这种形式常见于指定参数类型的时候。例如以下语句：

```
int getNumber(std::istream inputStream)
. . .
```

函数 getNumber 的 inputStream 参数是 istream 类型，istream 用的是 std 命名空间中的定义。在 std 命名空间定义的众多名称中，如类型名称 istream 是你唯一要用的(或者希望所有名称都采用 std::前缀进行类似限定)，就不需要使用以下语句：

```
using namespace std;
```

命名空间的微妙之处(选读)

“using 声明”(比如 using std::cout;)和“using 指令”(比如 using namespace std;)有两个区别，具体如下。

- using 声明只让命名空间中的一个名称进入可用状态，而 using 预编译指令使那个命名空间中的所有名称都进入可用状态。
- using 声明在代码中引入一个名称(比如 cout)，不允许这个名称再有其他用途。但 using 预编译指令只是隐式引入命名空间中的名称。只要代码中不实际使用冲突的名称，就不会出问题。

第一个区别很容易理解。第二个则不太容易。例如，假定命名空间 ns1 和 ns2 都提供了 myFunction 的定义，但除此之外再无其他名称冲突，那么只要永远不在代码中使用冲突名称 myFunction，以下语句就不会有任何问题：

```
using namespace ns1;
using namespace ns2;
```

相反，即使永远不使用 myFunction，以下语句也是非法的：

```
using ns1::myFunction;
using ns2::myFunction;
```

这个微妙之处有时会变得很重要，但大多数常规代码不需要关心这个问题。

自测题

10. 为 void 函数 wow 写函数声明。wow 函数有两个参数，第一个参数名为 s1，是 speed 类型，该类型在 speedway 命名空间中定义；第二个参数名为 s2，也是 speed 类型，但该类型在 indy500 命名空间中定义。
11. 图 11.4 展示了 Money 类的定义，其中包含以下函数声明：
```
void input(istream& ins);
void output(ostream& outs) const;
```
重写这些函数声明，删除它们之前的语句 ***using namespace*** std;(不需要回头看图 11.4)。

无名命名空间

图 12.1 和图 12.2 的 DigitalTime 类定义使用了三个辅助函数：digitToInt，readHour 和 readMinute。这些辅助函数是 ADT 类 DigitalTime“实现”的一部分，所以把这些函数的定义放到实现文件(图 12.2)。但这并不能将三个函数真正隐藏起来。我们希望这些函数对于 DigitalTime 类的实现文件来说是“局部”的。但基于目前的操作，它们并不是 DigitalTime 类的实现文件的局部函数。具体地说，在使用 DigitalTime 类的应用程序中，不能定义另一个名为 digitToInt(或 readHour 与 readMinute)的函数。这就违反了信息隐藏的原则。为真正隐藏这些辅助函数，将它们变成 DigitalTime 实现文件的局部函数，需将这些辅助函数放到特殊的**无名命名空间**(unnamed namespace)中。

编译单元(compilation unit)是一个文件，它可能是类的实现文件，以及通过#include 包含的其他所有文件，比如类的接口文件(头文件)。每个编译单元都有一个无名命名空间。为这种命名空间写命名空间分组时，要采取和其他任何命名空间一样的方式，只是不给出具体名称，如下所示：

```
namespace
{
    void sampleFunction()
       ...
} // 无名命名空间
```

无名命名空间定义的所有名称对编译单元来说都是局部的。因此，这些名称可在编译单元的外部重用于其他用途。例如，图 12.6 和图 12.7 展示了 DigitalTime 类的接口与实现文件的重写版本(也是最终版本)。注意，辅助函数(digitToInt，readHour 和 readMinute)都在无名命名空间中，所以它们相对于编译单元来说是局部的。如图 12.8 所示，无名命名空间定义的名称可在编译单元的外部用于其他目的。图 12.8 为应用程序的另一个函数重用了函数名 readHour。

分析图 12.7 的实现文件，发现辅助函数 digitToInt，readHour 和 readMinute 在无名命名空间外部使用时，没有添加任何命名空间限定符。在编译单元内的任何地方，无名命名空间中定义的所有名称都可不加限定地使用。当然，也只能如此，因为无名命名空间本来就没有可供限定的名称。

图 12.6　将整个类放到命名空间中：头文件

```
1  // 头文件 dtime.h：这是 DigitalTime 类的“接口”部分
2  // 这个类型的值表示一天当中的时间。值采取 24 小时制进行输入和输出
```

```
 3  // 例如，9:30 表示 9:30 AM，14:45 表示 2:45 PM
 4
 5  #ifndef DTIME_H
 6  #define DTIME_H
 7
 8  #include <iostream>
 9  using namespace std;
10
11  namespace dtimesavitch      <——  这是命名空间 dtimesavitch 的一个
12  {                                分组。命名空间 dtimesavitch 的另一
13                                   个分组在实现文件 dtime.cpp 中
14      class DigitalTime
15      {

16      <DigitalTime 类的定义与图 12.1 相同>
17      };
18  } // 结束 dtimesavitch
19
20  #endif // DTIME_H
```

图 12.7　将整个类放到命名空间中：实现文件

```
 1  // 实现文件 dtime.cpp(你的系统可能要求另一个不同于.cpp 的扩展名)：
 2  // 这是 ADT 类 DigitalTime 的“实现”部分
 3  // DigitalTime 类的接口放在头文件 dtime.h 中
 4  #include <iostream>
 5  #include <cctype>
 6  #include <cstdlib>
 7  #include "dtime.h"
 8  using namespace std;
 9
10  namespace   <——  无名命名空间的一个分组
11  {
12      // 以下函数声明在重载的输入操作符>>的定义中使用：
13
14      void readHour(istream& ins, int& theHour);
15      // 前条件：流 ins 中的下一个输入是一个采用 24 小时制的时间，比如 9:45 或 14:45
16      // 后条件：theHour 被设为时间的小时部分。冒号被舍弃，读取的下一个输入将是分钟数
17
18      void readMinute(istream& ins, int& theMinute);
19      // readHour 函数读取了小时数之后，这个函数从流 ins 中读取分钟数
20
21      int digitToInt(char c);
22      // 前条件：c 是'0'～'9'的一个数
23      // 返回数字的整数值；例如，digitToInt('3')返回 3
24  }// 无名命名空间
25
26
27  namespace dtimesavitch   <——  命名空间 dtimesavitch 的一个分组。另一个分组在头文件 dtime.h 中
28  {
29      bool operator ==(const DigitalTime& time1, const DigitalTime& time2)
        <重载操作符==剩余的定义与图 12.2 相同>
30
31      DigitalTime::DigitalTime()
        <这个构造函数剩余的定义与图 12.2 相同>
32      DigitalTime::DigitalTime(int theHour, int theMinute)
        <这个构造函数剩余的定义与图 12.2 相同>
33      void DigitalTime::advance(int minutesAdded)
        <这个 advance 函数剩余的定义与图 12.2 相同>
34
35      void DigitalTime::advance(int hoursAdded, int minutesAdded)
36      <这个 advance 函数剩余的定义与图 12.2 相同>
37
38      ostream& operator <<(ostream& outs, const DigitalTime& theObject)
        <重载操作符<<剩余的定义与图 12.2 相同>
39
40      // 使用 iostream 和无名命名空间中的函数：
41      istream& operator >>(istream& ins, DigitalTime& theObject)
```

```
42      {
43          readHour(ins, theObject.hour);
44          readMinute(ins, theObject.minute);
45          return ins;
46      }
47  } // dtimesavitch
48
49
50  namespace
51  {
52      int digitToInt(char c)
    <digitToInt 剩余的定义与图 12.2 相同>
53
54      void readMinute(istream& ins, int& theMinute)
    <readMinute 剩余的定义与图 12.2 相同>
55
56      void readHour(istream& ins, int& theHour)
    <readHour 剩余的定义与图 12.2 相同>
57
58  }// 无名命名空间
```

（注：无名命名空间定义的函数局部于这个编译单元(该文件和被 include 的其他文件)的。它们可在该文件的任何地方使用，但在该编译单元外没有意义）

（注：namespace ← 无名命名空间的另一个分组）

图 12.8　将整个类放到命名空间中——应用程序

```
1   // 这是应用程序文件 timedemo.cpp，演示了如何在一个无名命名空间中隐藏辅助函数
2   #include <iostream>
3   #include "dtime.h"
4
5
6   void readHour(int& theHour);
7
8   int main()
9   {
10      using namespace std;
11
12      using namespace dtimesavitch;
13
14      int theHour;
15      readHour(theHour);
16
17      DigitalTime clock(theHour, 0), oldClock;
18
19      oldClock = clock;
20      clock.advance(15);
21      if (clock == oldClock)
22          cout << "Something is wrong.";
23      cout << "You entered " << oldClock << endl;
24      cout << "15 minutes later the time will be "
25           << clock << endl;
26
27      clock.advance(2, 15);
28      cout << "2 hours and 15 minutes after that\n"
29           << "the time will be "
30           << clock << endl;
31
32      return 0;
33  }
34  void readHour(int& theHour)
35  {
36      using namespace std;
37
38      cout << "Let's play a time game.\n"
39           << "Let's pretend the hour has just changed.\n"
40           << "You may write midnight as either 0 or 24,\n"
41           << "but, I will always write it as 0.\n"
42           << "Enter the hour as a number (0 to 24): ";
43      cin >> theHour;
44      if (theHour == 24)
45          theHour = 0;
46  }
```

（注：第 5 行：将 using 预编译指令放在这里，程序行为不变）

（注：第 15 行：这个 readHour 不同于实现文件 dtime.cpp (图 12.7) 中的同名函数）

示范对话

```
Let's play a time game.
Let's pretend the hour has just changed.
You may write midnight as either 0 or 24,
but I will always write it as 0.
Enter the hour as a number (0 to 24): 11
You entered 11:00
15 minutes later the time will be 11:15
2 hours and 15 minutes after that
the time will be 13:30
```

一个有趣的问题是，无名命名空间如何满足以下 C++规则：“同一个命名空间中一个名称不能有两个定义”？每个编译单元都有一个无名命名空间，而不同编译单元很容易重叠。例如，一个类的实现文件和使用该类的应用程序通常都包含了类的头文件(接口文件或.h 文件)。所以，头文件就存在于两个编译单元中，同时参与了两个无名命名空间。虽然表面上危险，但只要每个编译单元的命名空间本身是有意义的，通常就不会产生任何问题。例如，假定在头文件的无名命名空间中定义了一个名称，那么在实现文件或应用程序文件的无名命名空间中，就不能再次定义它，这就避免了名称冲突。

无名命名空间

利用**无名命名空间**，可以使一个定义“局部”于编译单元(也就是文件及其包含的文件)。每个编译单元都有一个无名命名空间。无名命名空间中定义的每个标识符对于编译单元来说都是“局部”的。要将定义放到无名命名空间，将定义放到不指定具体名称的命名空间分组中即可，如下所示：

```
namespace
{
    Definition_1
    Definition_2
       ...
    Definition_Last
}
```

在编译单元的范围内，可直接使用无名命名空间中的任意名称，不需要使用限定符，完整示范程序参见图 12.6 和图 12.7。

■ 编程提示　为命名空间选择名称

在命名空间的名称中，最好包括你的姓氏或其他独一无二的字符串，以减小别人的命名空间和你的命名空间同名的可能性。如多个程序员为同一个项目写代码，务必确保不同命名空间具有不同名称，否则很容易就会在同一个作用域中出现同个名称的多个定义。正是由于这个原因，图 12.7 在命名空间名称 dtimesavitch 中使用了 savitch[①]一词。■

① 本书作者的姓。——译注

陷阱： 混淆全局命名空间和无名命名空间

不要混淆全局和无名命名空间。不将名称定义放到命名空间分组，默认在全局命名空间。要将名称定义放到无名命名空间，必须将其添加到一个命名空间分组，只是不指定具体命名空间名称，它以这些代码开头：

```
namespace
{
```

访问全局和无名命名空间中的名称都不需要限定符。但全局命名空间中的名称具有全局作用域(包括所有程序文件)，而无名命名空间中的名称是局部于一个编译单元的。

写代码时，一般不会混淆全局和无名命名空间，因为人们习惯于认为全局命名空间中的名称“没有命名空间”——即使这从技术上说并不正确。但讨论代码时，却很容易产生混淆。■

√ 自测题

12. 在图 12.8 的程序中，如果将以下 using 预编译指令：

```
using namespace dtimesavitch;
```

替换成以下 using 声明，程序的行为是否会改变？

```
using dtimesavitch::DigitalTime;
```

13. 以下程序输出什么？

```
#include <iostream>
using namespace std;

namespace sally
{
    void message();
}

namespace
{
    void message();
}

int main()
{
    {
        message();
        using sally::message;
        message();
    }

    message();

    return 0;
}

namespace sally
{
    void message()
    {
        cout << "Hello from Sally.\n";
    }
}

namespace
{
    void message()
    {
        cout << "Hello from unnamed.\n";
    }
}
```

14. 图 12.7 有两个无名命名空间分组：一个用于辅助函数的声明，一个用于辅助函数的定义。可以取消辅助函数声明的分组吗？如果可以，具体怎么做？

小　结

- 在 C++中，抽象数据类型(ADT)的所有成员变量都是私有的，它的各个操作实现为公共成员、非成员函数以及重载的操作符。
- 定义 ADT 类时，可将类的定义及其成员函数的实现放在独立的文件中。然后，ADT 类可独立于任何使用这个 ADT 类的程序进行编译。另外，同一个 ADT 类可以在任何数量的程序中使用。
- 命名空间是名称定义的集合，这些名称定义包括类定义和变量声明等。
- 可采取三种方式使用一个命名空间中的名称：“using 预编译指令”使命名空间中的所有名称都能使用；“using 声明”导入单独一个或几个名称；或使用命名空间的名称和作用域解析操作符直接限定名称。
- 要将定义放到命名空间，需将定义放到那个命名空间的分组中。
- 无名命名空间使名称定义局部于编译单元。

自测题答案

1. (a)，(b)和(c)放到接口文件中；(d)到(h)放到实现文件中(任何形式的 ADT 操作的定义都必须放到实现文件中)。(i)部分(也就是程序的 main 部分)要放到应用程序文件中。

2. 接口文件的名称以.h 结尾。

3. 只有实现文件需要编译。接口文件不需要。

4. 只有实现文件需要重新编译，但还需要重新链接文件。

5. 需要从图 12.1 的接口文件中删除私有成员变量 hour 和 minute，把它们替换成一个成员变量 minutes(注意加了一个 s)。除此之外，不需要在接口文件中进行其他任何修改。在实现文件中，需要修改所有构造函数和其他成员函数的定义，并修改重载的操作符的定义，使它们支持以新方式来记录时间(就本例来说，你不需要更改任何辅助函数，包括 readHour，readMinute 以及 digitToInt 等。但对于另一些类，甚至对于该类的另一些重新实现，可能需要更改这些辅助函数)。例如，重载的操作符>>的定义可更改为：

```
istream& operator >>(istream& ins, DigitalTime& theObject)
{
    int inputHour, inputMinute;
    readHour(ins, inputHour);
    readMinute(ins, inputMinute);
    theObject.minutes = inputMinute + (60 * inputHour);
    return ins;
}
```

   使用该类的任何应用程序文件都不必修改。但由于接口文件(和实现文件)已改变，所以需要重新编译任何应用程序文件。另外，当然需要重新编译实现文件。

6. 最简答案：ADT 是确保接口与实现进行良好分离的一个类。另外，将类描述成 ADT 时，我们认为非成员基本操作(比如重载的操作符)是 ADT 的一部分，即使技术上说不是 C++类的一部分。

7. 不可以。将 bigGreeting 替换成 greeting，表明在全局命名空间为 greeting 这个名称提供了一个定义。在程序的某些部分，命名空间 savitch1 的所有名称定义和全局命名空间的所有名称定义会出现共存的情况。在这些部分，**void** greeting();拥有两个不同的定义，所以会出错。

8. 可以，该附加定义不会产生任何问题。这是由于重载始终都是允许的。例如，在命名空间 savitch1 和全局命名空间都可用的时候，函数名称 greeting 会被重载。自测题 7 之所以有问题，是因为在某些时候，具有相同参数列表的函数名称 greeting 同时有两个不同的定义。

9. 可以，一个命名空间可拥有任意数量的分组。例如，图 12.5 为命名空间 savitch1 包含了两个分组：

```
namespace savitch1
{
    void greeting();
}

namespace savitch1
{
    void greeting()
    {
        cout << "Hello from namespace savitch1.\n";
    }
}
```

10. `void wow(speedway::speed s1, indy500::speed s2);`

11. `void input(std::istream& ins);`

```
void output(std::ostream& outs) const;
```

12. 程序的行为不会变化。

13.
```
Hello from unnamed.
Hello from Sally.
Hello from unnamed.
```

14. 可以。可取消用于辅助函数声明的分组，前提是辅助函数定义分组在辅助函数使用之前出现。例如，可删除辅助函数声明所在的命名空间，将辅助函数定义分组移到 dtimesavitch 命名空间分组(namespace dtimesavitch)之前。

编 程 练 习

编程练习一般只需写很小的程序，运用本章提到的编程概念。

1. 将以下成员函数添加到图 12.1 和图 12.2 定义的 ADT 类 DigitalTime 中：

```
void DigitalTime::intervalSince(const DigitalTime& aPreviousTime,
     int& hoursInInterval, int& minutesInInterval) const
```

该函数计算 DigitalTime 类型的两个值的间隔时间。DigitalTime 类型的一个值是调用成员函数 intervalSince 的对象，另一个值作为第一个参数提供。例如以下代码：

```
DigitalTime current(5, 45), previous(2, 30);
int hours, minutes;
current.intervalSince(previous, hours, minutes);
cout << "The time interval between " << previous
     << " and " << current << endl
     << "is " << hours << " hours and "
     << minutes << " minutes.\n";
```

在使用了 ADT 类 DigitalTime 修订版本的程序中，上述代码产生以下输出：

```
The time interval between 2:30 and 5:45
is 3 hours and 15 minutes.
```

第一个实参给定的时间可晚于调用对象的时间。这种情况下，我们认为第一个实参给定的时间是前一天的时间。你还应该写一个程序来测试这个修订过的 ADT 类。

2. 完整地做完自测题 5。写完整的 ADT 类，其中包括接口和实现文件。再写程序测试该 ADT 类。

3. 视频讲解：*Solution to Practice Program 12.3*

重做第 11 章的编程练习 1，用独立接口和实现文件定义 Money 这个 ADT 类，使它的实现能独立于任何应用程序进行编译。

4. 重做第 11 章的编程练习 2，用独立接口和实现文件定义 Pairs 这个 ADT 类，使它的实现能独立于任何应用程序进行编译。

5. 这个编程练习探究无名命名空间的工作原理。下面列出某个程序的一些片段，作用是对用户名和密码进行校验。用户名输入和校验代码放在一个文件中，密码输入和校验代码放在另一个文件中。

下面是 user.cpp 中的代码：

```
namespace Authenticate
{
   void inputUserName(0
   {
      do
      {
         cout << "Enter your username (8 letters only)" << endl;
         cin >> username;
      } while (!isValid());
```

```
    }
    string getUserName()
    {
        return username;
    }
}
```

在无名命名空间中定义 username 变量和 isValid()函数，使代码能通过编译。如 username 恰好包含 8 个字母，isValid()函数应返回 true。为上述代码生成一个恰当的头文件。

为 password.cpp 文件重复相同的步骤，将 password 变量和 isValid()函数放到无名命名空间中：

```
namespace Authenticate
{
    void inputPassword (0
    {
    do
    {
        cout << "Enter your password (at least 8 letters)" <<
            "and at least one non-letter)" << endl;
        cin >> password;
        } while (!isValid());
    }
    string getPassword ()
    {
        return password;
    }
}
```

对于密码，isValid()要求密码至少 8 个字符，而且其中至少有一个非字母字符。为这段代码也生成一个恰当的头文件。

现在有两个 isValid()函数，分别在不同的无名命名空间。将以下 main 函数放到合适的地方。这个程序应该编译并运行。

```
int main()
{
  inputUserName();
  inputPassword();
  cout <<   "Your username is " << getUserName() <<
            "and your password is: " <<
            getPassword() << endl;
  return 0;
}
```

用若干个非法的用户名和密码测试程序。

编 程 项 目

编程项目要求综合运用多方面的知识来解决问题，程序一般比编程练习大，解题方式多样化。

1. 重做第 11 章的编程练习 3。用独立的文件定义 ADT 类，使其能独立编译。
2. 重做第 11 章的编程项目 2。用独立的文件定义 ADT 类，使其能独立编译。
3. 重做第 11 章的编程项目 9。用独立的文件定义 ADT 类，使其能独立编译。将 main 函数放到它自己的文件中(独立于 ADT 文件)。

第 13 章　指针和链表

If somebody there chanced to be
Who loved me in a manner true
My heart would point him out to me
And I would point him out to you.

——*吉尔伯特和沙利文*[①]*，《鲁迪戈》*

概述

链表是用指针构造的列表。链表长度不固定，可在程序运行时伸缩。本章解释如何定义和操纵链表，并描述使用指针的一种新方式。

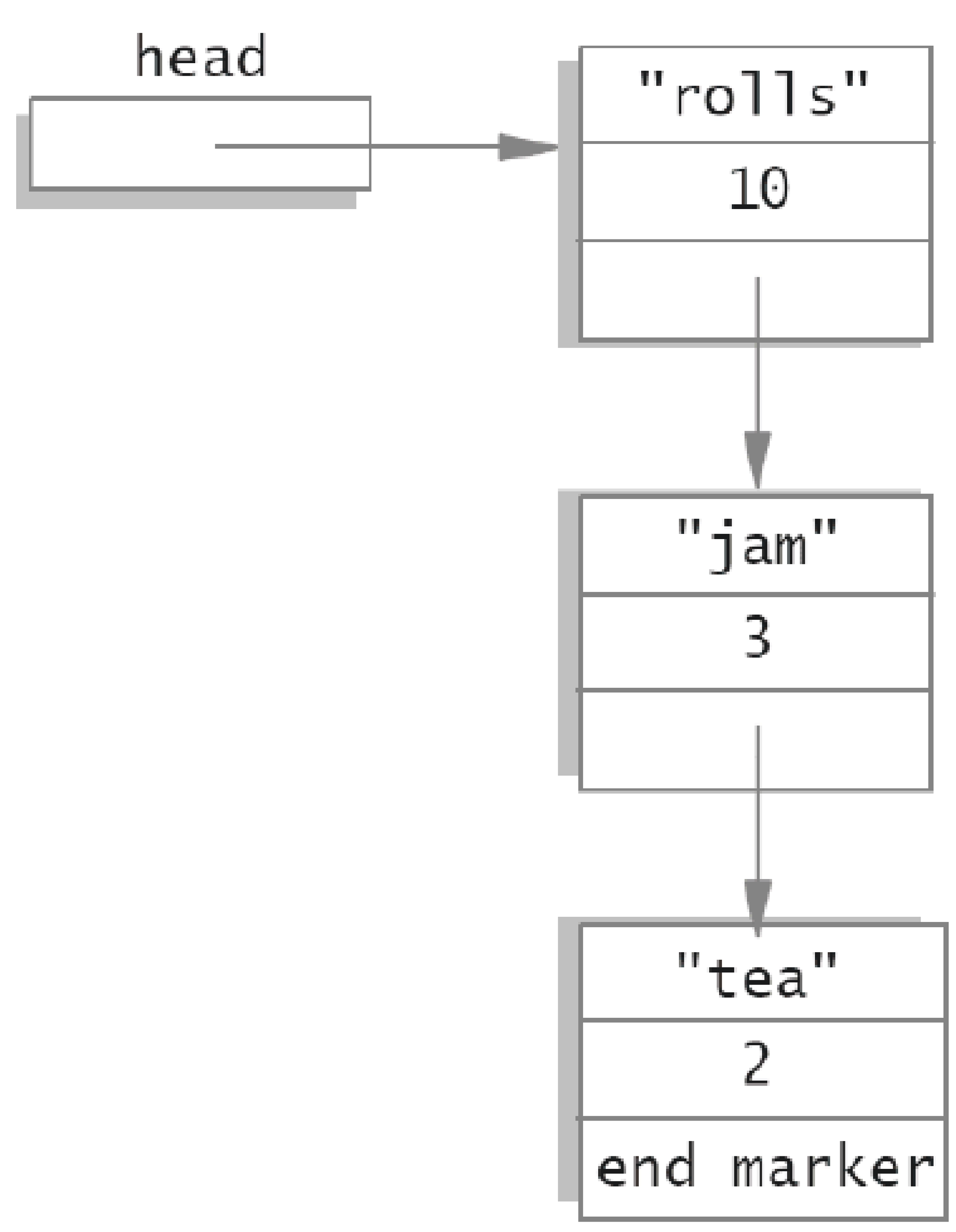
预备知识

本章基于第 2 章～第 12 章的知识。

13.1 节点和链表

真正有用的动态变量很少会是 int 或 double 这样的简单类型。相反，一般都是复杂类型，比如数组、结构或类。通过以前的学习，你体会到了数组类型的动态变量是多么有用。struct 或类类型的动态变量也很有用，只是方式不同。结构或类类型的动态变量通常有一个或多个成员变量，这些变量是指针，可将动态变量连接到其他动态变量。例如，图 13.1 展示了一个包含购物清单的结构。

图 13.1 节点和指针



① 吉尔伯特与沙利文(Gilbert and Sullivan)指的是英国维多利亚时代喜剧作家威廉 • S. 吉尔伯特(William S. Gilbert)与英国作曲家阿瑟 • 沙利文(Arthur Sullivan)的合作。从 1871 年到 1896 年长达二十五年的合作中，共同创作了 14 部喜剧。这里原文摘录《鲁迪戈》(Ruddigore)中的几句话。注意，有两个地方使用了“point to”。——译注

节点

图 13.1 的结构由通过箭头连接的方框构成。这些方框称为**节点**(node)，箭头代表指针。图 13.1 的每个节点都包含一个字符串、一个整数以及一个指向同类型的其他节点的指针。注意指针指向的是整个节点，而不是指向节点内部的单独数据项(比如 10 或者"rolls")。

在 C++中，节点作为结构或类来实现。例如，以下代码定义了在图 13.1 中代表节点的 struct 类型，并定义了指向这种节点的指针类型：

```
struct ListNode
{
    string item;
    int count;
    ListNode *link;
};
typedef ListNode* ListNodePtr;
```

类型定义的顺序很重要。ListNode 必须先定义，因为它要在 ListNodePtr 的定义中使用。

在图 13.1 中，标记为 head(表头)的框不是节点，而是能指向节点的指针变量。指针变量 head 像下面这样声明：

```
ListNodePtr head;
```

虽然排列了类型定义的顺序，避免出现某些非法的循环定义，但很明显，ListNode 结构类型的定义仍然是循环的：在结构定义内部，用类型名称 ListNode 定义成员变量 link。这种循环定义没有问题，在 C++中是允许的。定义在逻辑上讲得通，一个证据便是你能描绘出如图 13.1 所示的结构图。

现在已在 struct 内部定义了指针，而且让这些指针指向其他包含指针的 struct。虽然语法让人感觉很“纠结”，但在所有情况下，语法都符合我们为指针和 struct 描述的那几条规则。以图 13.1 的情况为例，假定像前面那样声明，而且希望将第一个节点中的数字从 10 变成 12。一个办法是使用以下语句：

```
(*head).count = 12;
```

赋值操作符左侧的表达式需稍作解释。变量 head 是指针变量。所以，表达式*head 是它所指向的东西，也就是包含"rolls"和整数 10 的那个节点(动态变量)。*head 引用的节点是一个 struct，而这个 struct 的成员变量(包含 int 类型的值)叫 count。因此，(*head).count 是第一个节点中的 int 变量的名称。*head 两侧的圆括号不是可有可无的。你希望提领操作符(*)先于圆点操作符(.)执行。但圆点操作符的优先级高于提领操作符*。所以，假如没有圆括号，首先执行的就是圆点操作符，这会产生一个错误。下面将介绍一种助记符号，它能避免你对圆括号的担心。

C++支持**箭头操作符**(arrow operator)->，指定由指针指向的 struct 或类的成员。箭头操作符->合并了提领操作符*和圆点操作符的功能，表示箭头左侧的指针指向右侧的动态 struct 或对象的成员。例如，上述赋值语句(用于更改第一个节点中的数字)可简化为：

```
head->count = 12;
```

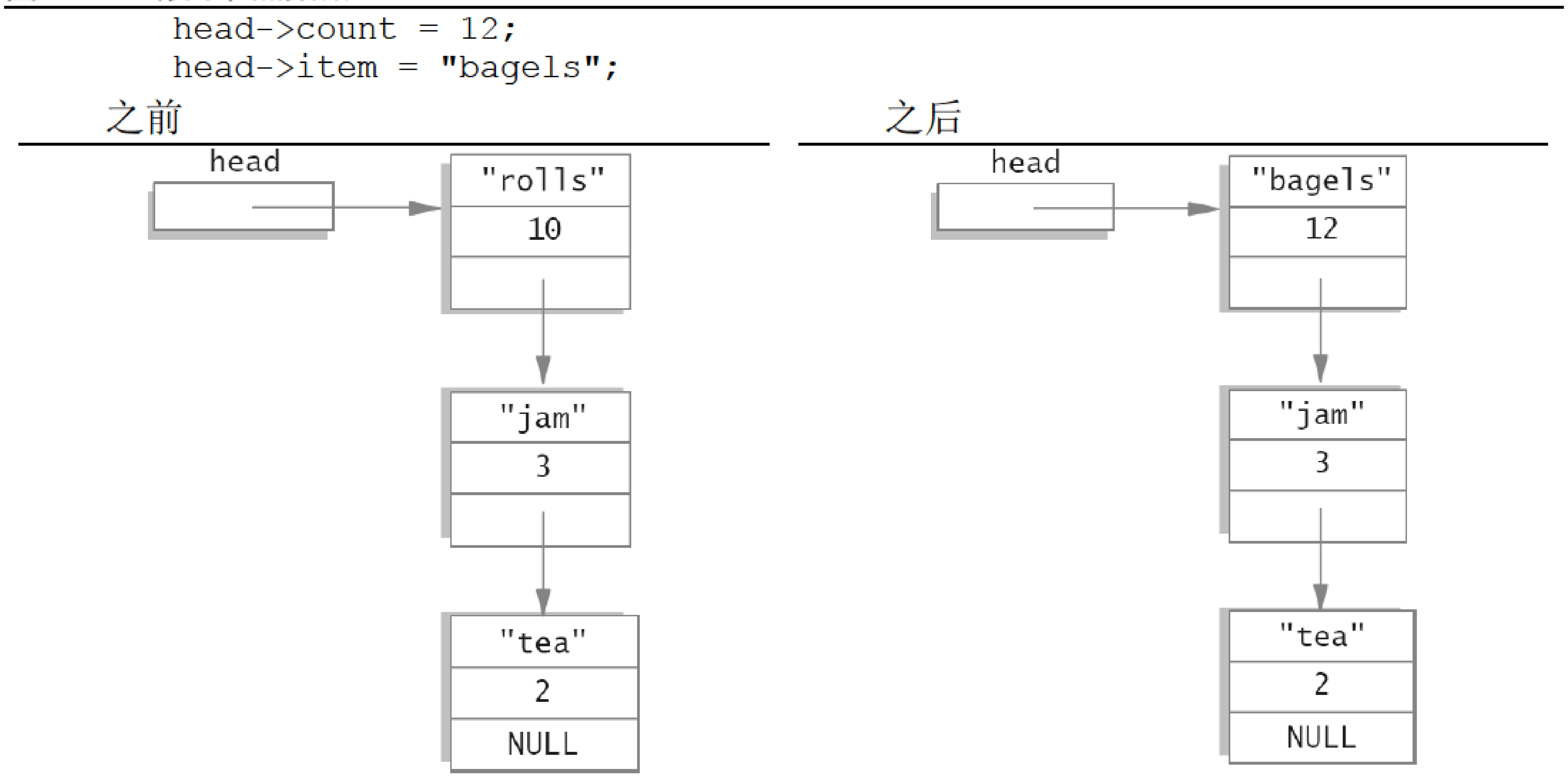
两个赋值语句的效果一样，但这种形式最常用。

使用以下语句，可将第一个节点中的字符串从"rolls"变成"bagels"：

```
head->item = "bagels";
```

图 13.2 展示了对列表第一个节点进行更改的结果。

图 13.2　访问节点数据



箭头操作符->

箭头操作符->指定由一个指针变量指向的 struct (或类的对象)的成员。语法如下：

```
Pointer_Variable->Member_Name
```

上述语句引用 struct 或对象的一个成员，*Pointer_Variable* 指向这个成员。具体引用的成员由 *Member_Name* 指定。例如，给定以下定义：

```
struct Record
{
    int number;
    char grade;
};
```

以下语句创建 Record 类型的动态变量，将动态 struct 变量的成员变量设为 2001 和'A'：

```
Record *p;
p = new Record;
p->number = 2001;
p->grade = 'A';
```

下面分析图 13.2 的列表中的最后一个节点中的指针成员。最后一个节点在本来是指针的位置标注了单词 NULL。在图 13.1 中，我们在相同位置标注的是“end marker”(结束标记)。但是，“end marker”不是有效的 C++表达式。在 C++程序中，使用常量 NULL 标记列表结束。NULL 是特殊的已定义常量，它是 C++语言的一部分(作为 C++库的必要部分提供)。

NULL 通常用于两个不同(但又通常一致)的目的。可用它向一个指针变量赋值(不然指针变量将不包含任何值)。这样可防止因为疏忽而引用一个真实存在的内存位置，因为 NULL 不是任何内存位置的地址。第二种用法就是作为结束标记。程序可遍历如图 13.2 所示的列表。一旦遇到包含 NULL 的节点，就知道抵达列表尾部。

常量 NULL 实际是数字 0，但我们宁愿将其视为 NULL，并拼写为 NULL。这样可清楚地指出是要将这个特殊用途的值赋给指针变量。标识符 NULL 的定义存在于大量标准库中，比如<iostream>和<cstddef>。所以，要使用 NULL，必须用 include 指令来包含<iostream>或<cstddef>(或其他合适的库)。但为了在程序代码中使用 NULL，你不需要再添加一个 using 指令。具体地说，只是想使用 NULL 就不必添加 *using namespace* std; (当然，代码中的其他操作可能需要像 *using namespace* std;这样的语句)。[①]

指针可用赋值操作符设为 NULL。下面声明名为 there 的指针变量，并初始化为 NULL：

```
double *there = NULL;
```

常量 NULL 可赋给任意指针类型的指针变量。

> **NULL**
>
> NULL 是特殊常量值，向否则没有值的指针变量赋值。可将 NULL 赋给任意类型的指针变量。许多库都定义了标识符 NULL，其中包括头文件为<cstddef>的库和头文件为<iostream>的库。常量 NULL 实际是数字 0，但我们宁愿把它视为 NULL，并拼写为 NULL。

nullptr

常量 NULL 实际是数字 0，这造成了歧义，例如以下重载的函数：

```
void func(int *p);
void func(int i);
```

func(NULL)实际调用哪个函数？由于 NULL 是数字 0，所以两者均有效。C++11 的解决方案是引入新常量 nullptr。nullptr 不是整数零，而是代表空指针的字面值常量。以前为指针使用 NULL 的所有地方都改成使用 nullptr，例如以下代码：

```
double *there = nullptr;
```

> **nullptr**
>
> nullptr 是用法和 NULL 一样的常量值，但*只能*赋给指针。它不是数字 0。用 nullptr 区分空指针和数字 0。nullptr 在 C++11 中引入。

√ 自测题

1. 假定程序包含以下类型定义：

① NULL 的定义由 C++预处理器控制，它会将 NULL 替换成 0。因此，编译器从来不会真正看到“NULL”，这样就不存在命名空间的问题，所以不需要 using 预编译指令。

```
struct Box
{
    string name;
    int number;
    Box *next;
};
typedef Box* BoxPtr;
```

以下代码输出什么？

```
BoxPtr head;
head = new Box;
head->name = "Sally";
head->number = 18;
cout << (*head).name << endl;
cout << head->name << endl;
cout << (*head).number << endl;
cout << head->number << endl;
```

2. 假定程序包含如自测题 1 所示的类型定义和代码。那些代码会创建一个节点，其中包含 string 值 "sally"和数值 18。应该添加什么代码，才能将该节点的成员变量 next 的值设为 NULL？
3. 假定程序包含如自测题 1 所示的类型定义和代码。假定指针变量 head 的值没有改变，如何销毁 head 指向的动态变量，将它使用的内存归还给自由存储，以便在创建新的动态变量时重用那些内存？
4. 给定以下结构定义：

```
struct ListNode
{
    string item;
    int count;
    ListNode *link;
};
ListNode *head = new ListNode;
```

请编写代码，将字符串值"Wilbur's brother Orville"赋给 head 指向的节点的 item 成员。

链表

如图 13.2 所示的列表称为**链表**(linked list)。链表是节点列表，每个节点都有一个指针成员变量指向列表中的下一个节点。第一个节点称为**表头**(head)，所以，将指向第一个节点的指针变量命名为 head。注意名为 head 的指针本身不是表头，它只是指向表头。最后一个节点没有特别名称，但有一项特殊属性。最后一个节点将 NULL 用作它的成员指针变量的值。要检查一个节点是不是最后一个节点，检查该节点的指针变量是否等于 NULL 就可以了。

本节目标是写一些基本函数来操纵链表。考虑到例子的多样性，并简化标注，我们使用了比图 13.2 的节点更简单的一种类型。这些节点只包含一个整数和一个指针。下面是准备使用的节点和指针类型定义：

```
struct Node
{
    int data;
    Node *link;
};

typedef Node* NodePtr;
```

作为热身练习，下面分析一下如何构造具有这种节点类型的一个链表的开头部分。首先要声明一个指针变量，把它命名为 head，它指向链表的表头：

```
NodePtr head;
```

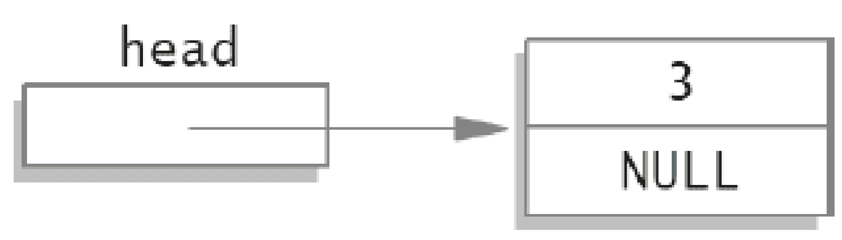
为了创建第一个节点，要用 new 操作符新建动态变量，它将成为链表的第一个节点：

```
head = new Node;
```

接着，向新节点的成员变量赋值：

```
head->data = 3;
head->link = NULL;
```

注意，该节点的指针成员等于 NULL。这是由于就现阶段来说，该节点既是列表的第一个节点，也是最后一个。现阶段链表如下所示。



这个单节点列表纯粹以人工方式构造。要获得更大的链表，程序必须能以一种系统化的方式添加节点。下面将简单介绍如何在链表中插入节点。

在表头插入节点

假定链表已包含一个或更多节点，现在要开发函数来插入另一个节点。函数第一个参数是传引用参数，代表指向链表表头的指针变量(也就是指向链表第一个节点的指针变量)。另一个参数指定要在新节点中存储的数字。这个插入函数的函数声明如下：

```
void headInsert(NodePtr& head, int theNumber);
```

> **链表作为实参**
>
> 始终都要有一个指向链表表头的指针变量。我们借助该指针变量来命名链表。如函数获取链表作为实参，该指针(它指向链表表头)就可作为链表实参使用。

为了在链表中插入新节点，函数用 new 操作符创建新节点。然后将数据复制到新节点，并在表头位置插入新节点。像这样插入节点，新节点将成为链表的第一个节点(表头节点)，而不是最后一个节点。由于动态变量没有名称，必须让一个局部指针变量指向该节点。将局部指针变量命名为 tempPtr，新节点就可用*tempPtr 来引用。完整过程总结如下。

headInsert 函数的伪代码：

1. 新建由 tempPtr 指向的动态变量(新动态变量就是新节点，用*tempPtr 引用)
2. 将数据放到新节点中
3. 使新节点的 link 成员指向原始链表的表头节点(第一个节点)
4. 让指针变量 head 指向新节点

图 13.3 展示了该算法，图中的步骤 2 和步骤 3(伪代码的步骤 3 和步骤 4)用以下 C++ 赋值语句表示：

```
tempPtr->link = head;
head = tempPtr;
```

完整函数定义参见图 13.4。

图 13.3　在链表中添加节点

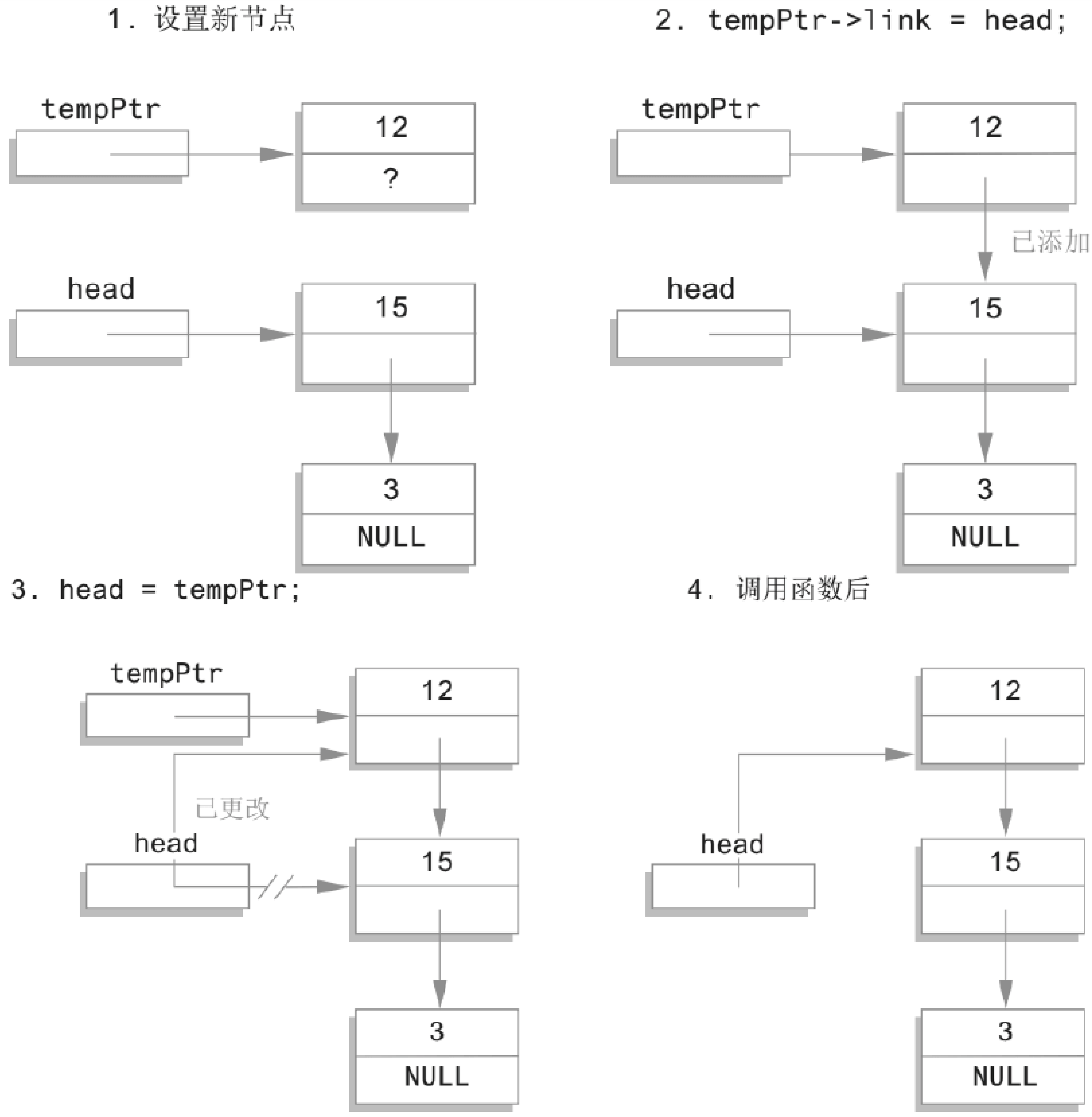


图 13.4　在链表表头添加节点的函数

函数声明

```
1  struct Node
2  {
3      int data;
4      Node *link;
5  };
6
7  typedef Node* NodePtr;
8
9  void headInsert(NodePtr& head, int theNumber);
10 // 前条件：指针变量 head 指向一个链表的表头
11 // 后条件：在链表的表头，添加一个包含 theNumber 的新节点
```

函数定义

```
1  void headInsert(NodePtr& head, int theNumber)
2  {
3      NodePtr tempPtr;
4      tempPtr = new Node;
5
6      tempPtr->data = theNumber;
7
8      tempPtr->link = head;
9      head = tempPtr;
10 }
```

要考虑列表不含任何内容的可能性。例如，本周不采购，购物清单应该为空。什么都没有的列表称为**空列表**(empty list)。命名链表需命名指向表头的指针，但空列表没有表头节

点。我们用指针 NULL 指定空列表。假定指针变量 head 本应指向链表的表头节点，但你希望指出列表为空，可以像这样设置 head 值：

```
head = NULL;
```

设计用于操纵链表的函数时，必须核实它能否处理空列表。不能就需考虑添加针对空列表的特例。如果不能设计函数来支持空列表，程序必须能以其他方式处理空列表，或彻底避免出现空列表。幸好，一般都可以像对待其他任何列表那样对待空列表。例如，图 13.4 的 headInsert 函数是面向非空列表而设计的，但稍微研究一下，就会发现它同样适合空列表。

陷阱：丢失节点

写 **headInsert** 函数的定义时，你可能倾向于使用指针变量 head 而不是局部指针变量 tempPtr 来构造新节点。所以，函数定义可能像下面这样开头：

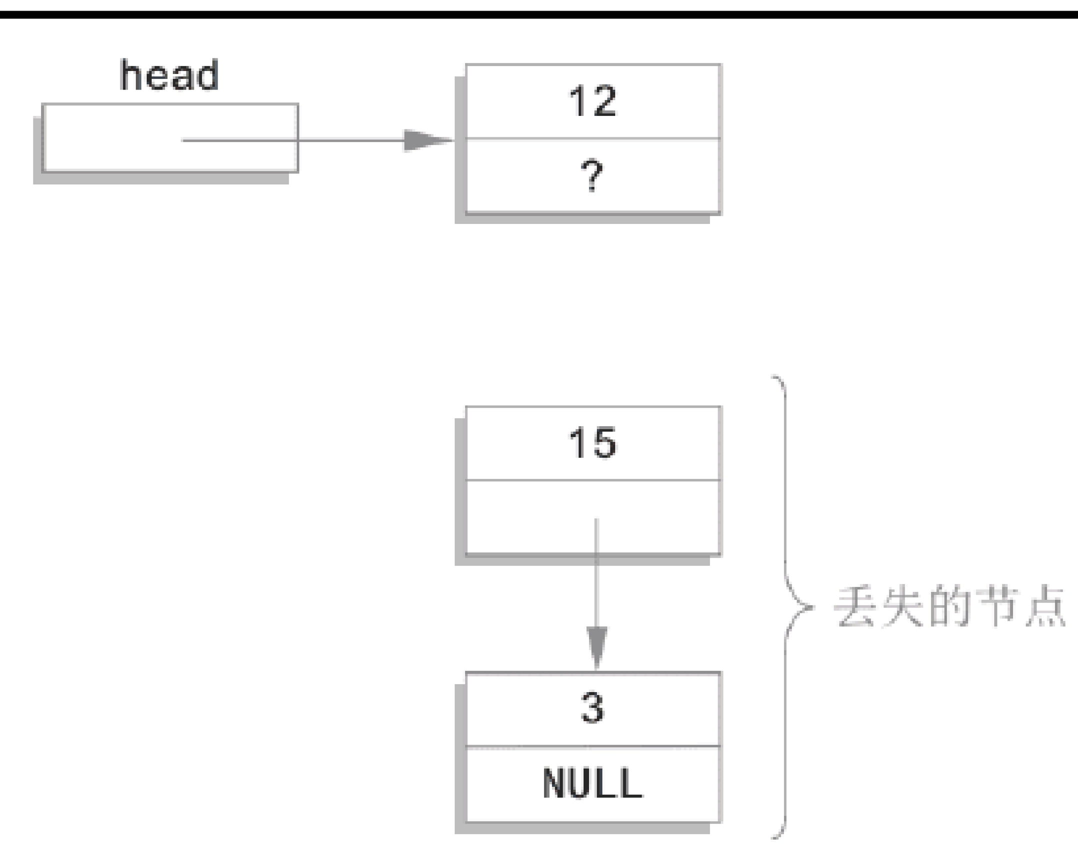
```
head = new Node;
head->data = theNumber;
```

至此，新节点确实能成功构造，它会包含正确的数据，而且会由指针 head 来指向，似乎一切正常。剩下的工作便是设置以下指针成员，使其指向列表原先的第一个节点，从而将列表剩余部分连接到该节点：

```
head->link
```

图 13.5 展示了新数据值为 12 时的情形，它清楚揭示了其中存在的问题。像这样操作，就没有任何东西指向包含 15 的节点。由于没有一个已命名的指针指向它(或指向以那个节点开头的一个指针链)，所以程序无法引用该节点。该节点下方的节点也会丢失。程序无法用指针来指向这两个节点，无法访问这些节点中的数据，也无法对这些节点采取其他任何操作。总之，无法引用这些节点。

图 13.5　丢失的节点



只要程序在运行，这种情况就会一直持续。如程序含有丢失的节点，就可以说它存在**内存泄漏**。内存泄漏太严重，会导致程序耗尽内存，最终异常终止。更糟的是，普通用户程序中的内存泄漏(丢失的节点)可能导致整个操作系统崩溃。为避免产生这种丢失的节点，程序必须始终保持一个指向表头的指针，通常是 head 这样的一个指针变量中的指针。　■

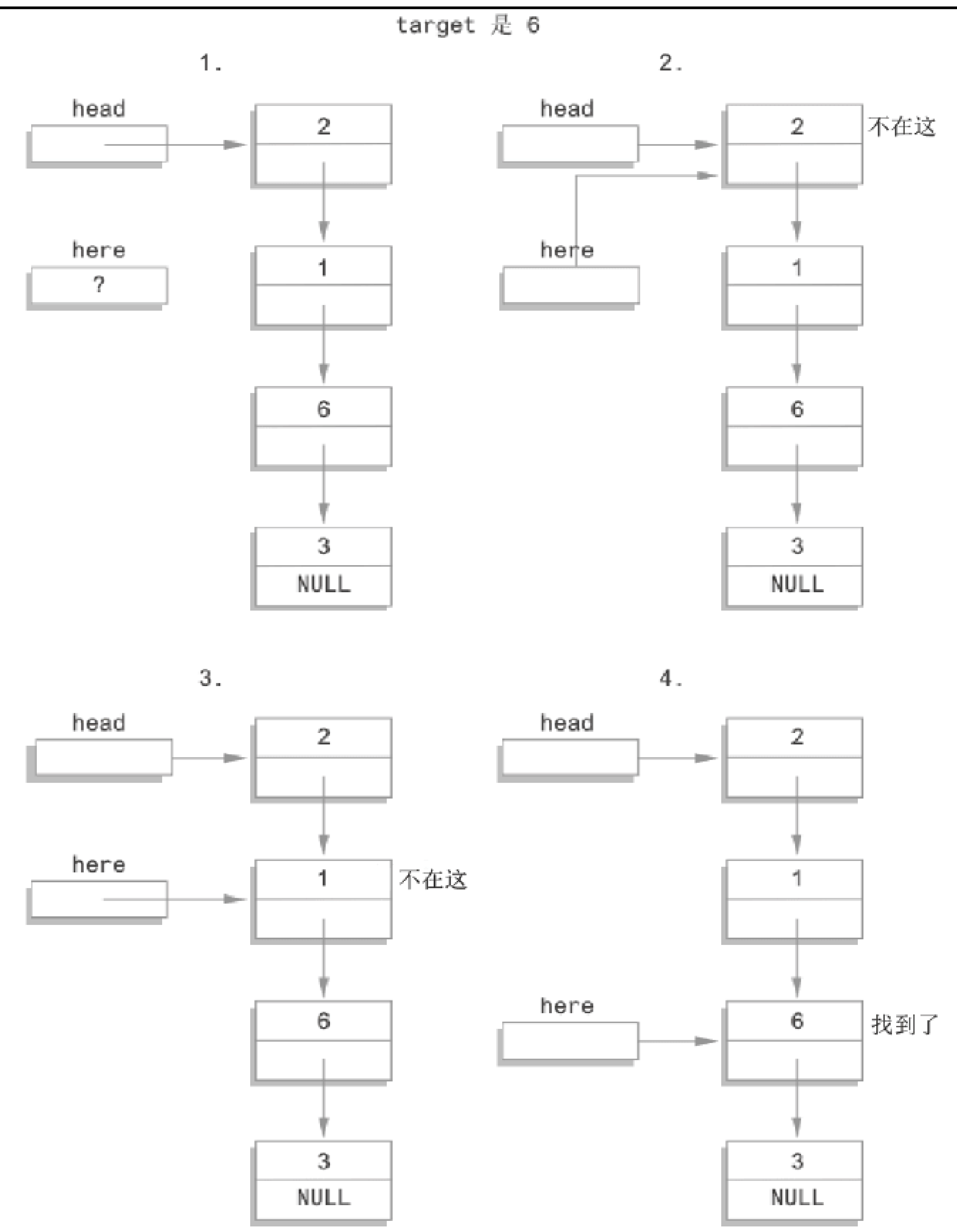
搜索链表

接着设计函数来搜索链表以定位特定节点。准备使用和之前一样的节点类型，也就是 Node(图 13.4 给出了节点和指针类型的定义)。该函数应该有两个参数：链表和想要定位的整数。函数应返回一个指针，指向包含那个整数的第一个节点。如果没有任何节点包含指定整数，函数返回指针 NULL。这样程序就能检查函数返回的指针值是否等于 NULL，从而判断整数是否在列表中。函数声明和函数头的注释文字如下所示：

```
NodePtr search(NodePtr head, int target);
// 前条件：head 指针指向链表的表头
// 最后一个节点的指针变量是 NULL
// 如果列表为空，那么 head 为 NULL，返回一个指针，指向包含 target 的第一个节点
// 如果没有任何节点包含 target，函数返回 NULL
```

我们准备用名为 here 的局部指针变量遍历列表，并在遍历过程中查找 target。要遍历列表(或由节点和指针构成的其他任何数据结构)，唯一的办法就是跟随指针。所以，先让 here 指向第一个节点，并跟随每个节点的指针，将指针从一个节点移到另一个。图 13.6 展示了这种技术。由于空列表会带来一些小问题，可能干扰我们的讨论，所以刚开始假定链表至少包含一个节点。以后还会重拾这个话题，检查算法是否适合空列表。

图 13.6　搜索链表



下面是这种搜索技术的算法陈述。

search 函数的伪代码

```
让指针变量 here 指向链表的表头节点(第一个节点)。
while (here 指向的节点不包含 target，而且 here 指向的不是最后一个节点)
{
    使 here 指向列表的下一个节点
}
if (here 指向的节点包含 target)
    return here;
else
    return NULL;
```

指针 here 要移至下一个节点，必须依赖当前可用的已命名指针。下一个节点由 here 当前指向节点的指针成员来指向。here 当前指向节点的指针成员由以下表达式给出：

```
here->link
```

here 要移至下一个节点，需更改 here，使其指向上述已命名指针(成员)变量指向的节点。所以，以下语句将指针 here 移至列表下一个节点：

```
here = here->link;
```

综上所述，算法伪代码修订如下：

search 函数代码的初步版本

```
here = head;
while (here->data != target && here->link != NULL)
        here = here->link;
if (here->data == target)
    return here;
else
    return NULL;
```

注意 while 语句中的布尔表达式。我们要检查成员变量 here->link 是否不等于 NULL，从而测试 here 是否不指向最后一个节点。

还必须证实一下它对空列表的处理。检查上述代码，发现当它遇到空列表时，会产生一个问题。如果是空列表，here 就等于 NULL，所以以下表达式是未定义的：

```
here->data
here->link
```

其中的 here 等于 NULL，不指向任何节点，所以既没有名为 data 的成员，也没有名为 link 的成员。因此，需要为空列表准备一个特殊情况。完整函数定义参见图 13.7。

图 13.7　在链表中定位节点的函数

函数声明

```
1  struct Node
2  {
3      int data;
4      Node *link;
5  };
6
7  typedef Node* NodePtr;
8
9  NodePtr search(NodePtr head, int target);
```

```
10  // 前条件：head 指针指向链表的表头
11  // 最后一个节点的指针变量是 NULL
12  // 如果列表为空，那么 head 为 NULL，返回一个指针，它指向包含 target 的第一个节点
13  // 如果没有任何节点包含 target，函数返回 NULL。
```

函数定义

```
1  // 使用 cstddef:
2  NodePtr search(NodePtr head, int target)
3  {
4      NodePtr here = head;
5
6      if (here == NULL)
7      {                              处理空列表
8          return NULL;
9      }
10     else
11     {
12         while (here->data != target && here->link != NULL)
13             here = here->link;
14
15         if (here->data == target)
16             return here;
17         else
18             return NULL;
19     }
20 }
```

指针作为迭代器

迭代器(iterator)允许遍历数据结构中存储的数据项，针对每个数据项执行指定操作。迭代器可以是某个迭代器类的对象，或者是某种更简单的东西，比如数组索引或指针。指针提供了迭代器的简单例子。事实上，可将指针视为迭代器的原型。以链表为例可以很容易地理解迭代器的基本思路。将指针用作迭代器，从表头开始，在链表中每次移经一个节点，最终遍历链表的所有节点。常规形式如下：

```
Node_Type *iter;
for (iter = head; iter != NULL; iter = iter->link)
    对 iter 指向的节点做你想做的事情;
```

其中，*head* 是指向链表表头节点的指针，*link* 是节点中指向下个节点的成员变量的名称。

例如，要输出我们讨论的那个链表的所有节点的数据，可考虑使用以下代码：

```
NodePtr iter; // 等价于：Node *iter;
for (iter = head; iter != NULL; iter = iter->Link)
    cout << (iter->data);
```

Node 和 NodePtr 的定义参见前面的图 13.7。

在列表中插入和删除节点

接下来设计一个函数，它能在链表任何指定位置插入节点。如希望节点按特定顺序排列，比如按数字或字母顺序，就不能简单地在链表开头或结尾插入节点。所以要设计函数在链表指定节点之后插入节点。假定另一个函数或者程序的另一个部分已正确建立了名为 afterMe 的指针，它指向链表的某个节点。现在，我们希望将新节点放在 afterMe 指向的那个节点之后，如图 13.8 所示。同样的技术适合包含任何数据的节点，但为了使讨论具体化，我们使用的节点与前几节讨论的节点具有相同类型。类型定义已在图 13.7 中给出。我

们想定义的函数的声明如下：

```
void insert(NodePtr afterMe, int theNumber);
// 前条件：afterMe 指向链表中的一个节点
// 后条件：将包含 theNumber 的一个新节点放在 afterMe 指向的节点之后
```

新节点的建立方式与图 13.4 的 headInsert 函数一样。区别在于，现在不是在表头插入节点，而是在 afterMe 指向的节点之后插入。图 13.8 展示了这种插入方式，并可用以下 C++代码表示：

```
// 添加从新节点到列表的链接:
tempPtr->link = afterMe->link;
// 添加从列表到新节点的链接:
afterMe->link = tempPtr;
```

两个赋值语句的顺序很关键。在第一个赋值语句中，希望获得 afterMe->link 发生改变之前的指针值。完整函数在图 13.9 中给出。

分析 insert 函数的代码，会发现即使 afterMe 指向的是列表最后一个节点，函数也能正常工作。但 insert 不能在链表起始处插入节点。要在列表开头插入节点，可使用图 13.4 给出的 headInsert 函数。

使用 insert 函数，可按照数字顺序、字母顺序或其他顺序对链表进行排序。要在某个位置“挤”入一个新节点，只需调节两个指针——不管链表有多长，也不管要在列表的什么位置插入新数据。相反，如换用数组，就必须复制数组中的大部分(极端情况下甚至要复制全部)，才能在正确位置为一个新值留出空间。虽然在定位指针 afterMe 时会产生一定开销，但在链表中插入时，效率通常都高于在数组中插入。

图 13.8　在链表中部插入

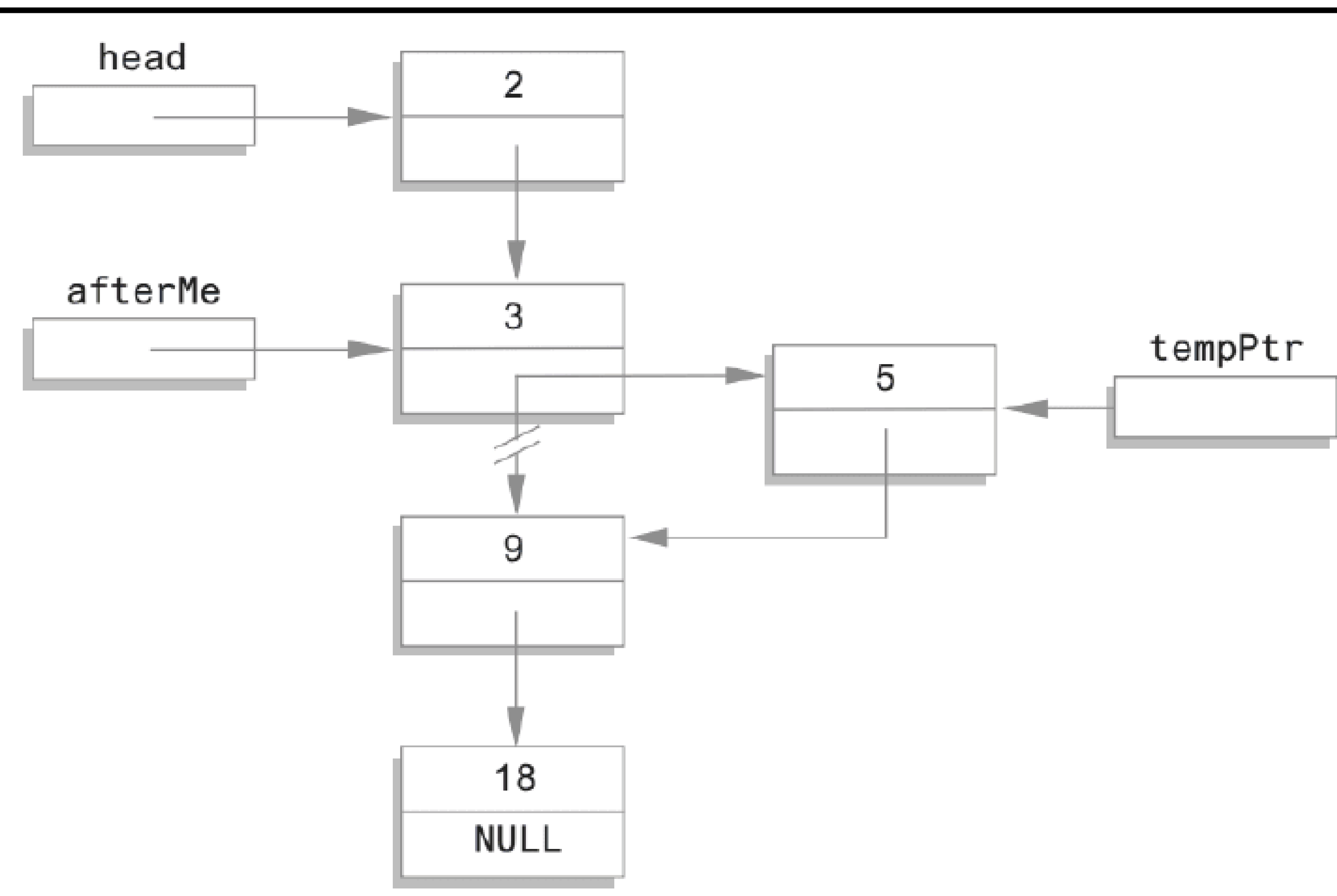


图 13.9　在链表中部添加节点的函数

函数声明

```
1  struct Node
2  {
3      int data;
4      Node *link;
5  };
```

```
 6
 7  typedef Node* NodePtr;
 8
 9  void insert(NodePtr afterMe, int theNumber);
10  // 前条件：afterMe 指向链表中的一个节点
11  // 后条件：在 afterMe 指向的节点之后，添加一个包含 theNumber 的新节点
```

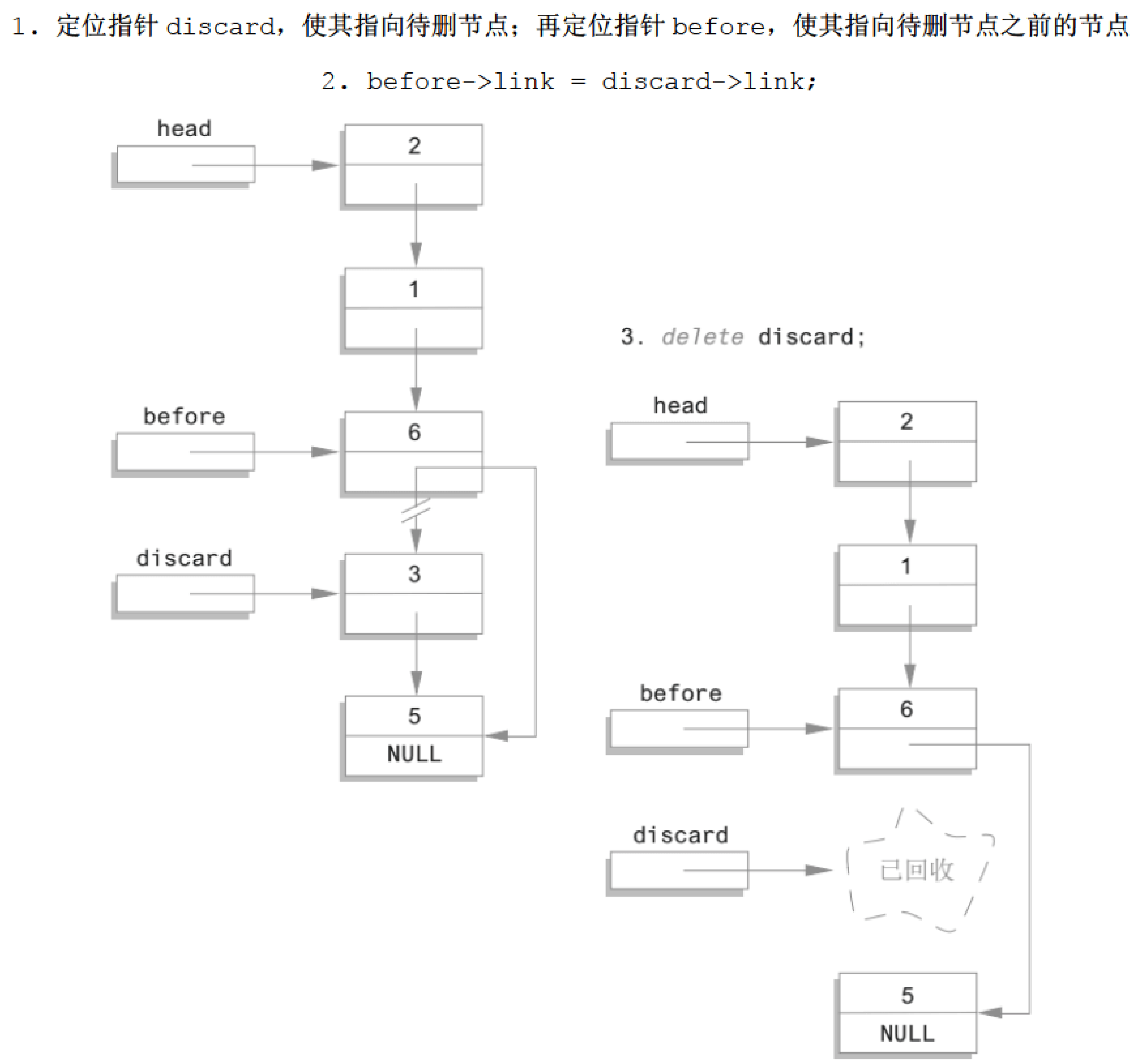
函数定义

```
 1  void insert(NodePtr afterMe, int theNumber)
 2  {
 3     NodePtr tempPtr;
 4     tempPtr = new Node;
 5
 6     tempPtr->data = theNumber;
 7
 8     tempPtr->link = afterMe->link;
 9     afterMe->link = tempPtr;
10  }
```

从链表删除节点也很容易。图 13.10 展示了具体过程。一旦定位好指针 before 和 discard，执行以下语句即可删除节点：

```
before->link = discard->link;
```

图 13.10　删除节点



只是从链表中删除节点，上述语句足够。但如果该节点不再使用，就应销毁它，以便自由存储(堆)回收它占用的内存。为此，可以这样调用 delete：

```
delete discard;
```

陷阱：为动态数据结构使用赋值操作符

假定 head1 和 head2 是指针变量，而且 head1 指向链表表头节点，以下语句使 head2 指向同一个表头节点。换言之，指向同一个链表：

```
head2 = head1;
```

但是必须记住一点：事实上只存在一个链表，而不是两个。所以，一旦更改了 head1 指向的链表，就同时更改了 head2 指向的链表，因为它们指向同一个链表。

如 head1 指向某个链表，而你希望 head2 指向该链表的一个完全相同的拷贝，上述赋值语句将无法获得你想要的效果。相反，必须逐个节点复制整个链表。另外，还可重载赋值操作符=来支持你想要的操作。对操作符=的重载已在第 11 章的“重载赋值操作符”一节进行了讨论。 ■

√ 自测题

5. 为一个链表中的节点和指针编写类型定义。节点类型命名为 NodeType，指针类型命名为 PointerType。链表作为字母列表使用。
6. 为了给出一个链表，通常需要让一个指针指向列表的第一个节点。但空列表没有第一个节点。在这种情况下，通常用什么指针值来表示空列表？
7. 假定程序包含以下类型定义和指针变量声明：

```
struct Node
{
    double data;
    Node *next;
};

typedef Node* Pointer;
Pointer p1, p2;
```

   假定 p1 指向某个链表的一个上述类型的节点。编写代码，让 p1 指向该链表的下一个节点(指针 p2 下一题用，和本题无关)。

8. 假定程序包含如自测题 7 所示的类型定义和指针变量声明。进一步假定 p2 指向上述类型的一个节点，该节点在一个链表中，而且不是该链表的最后一个节点。写代码删除 p2 指向的节点之后的节点。执行代码后，链表中只是少了一个节点，没有别的变化。**提示：**可能要另外声明一个指针变量。
9. 选择正确答案并解释。假定有一个大型数组和一个大型链表，其中含有相同类型的对象。与数组相比，在链表中部的一个已知位置插入新对象时，效率的情况会如何？
   a. 效率更高
   b. 效率更低
   c. 效率基本相同
   d. 效率视两者长度而定

链表的变体

本节解释如何使用节点和指针创建多种数据结构。将简单描述另外两个数据结构，即双链表和二叉树。

普通链表只允许朝一个方向，在列表中向下移动(跟随链接)。相反，**双链表**(doubly linked list)有两个链接，一个指向下一个节点，另一个指向上一个，如图 13.11 所示。双链表的节点类可以像下面这样定义：

```
struct Node
{
    int data;
    Node *forwardLink;
    Node *backLink;
};
```

双链表通常各有一个指针指向表头和表尾结点。相反，普通链表只有一个指针指向表头节点。可将两个指针称为 forward 和 back，但谁是表头谁是表尾可任意决定。双链表类的构造函数和部分函数的定义必须在单链表的基础上进行修改，以适应额外的链接。

图 13.11　双链表

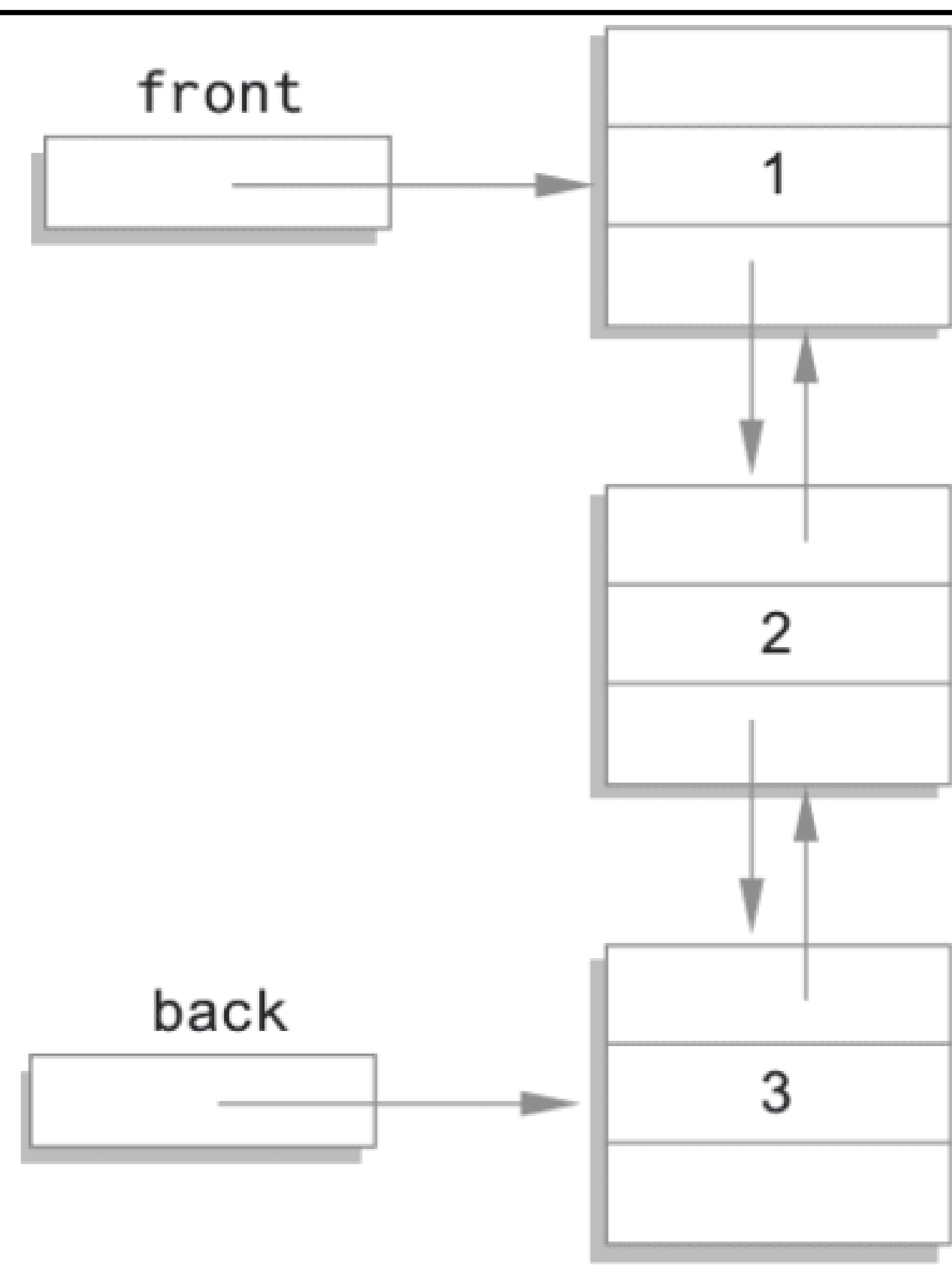


图 13.12 展示了称为**树**的特殊数据结构。具体地说，在树中可沿着某个路径跟随链接，从顶部(根)节点抵达任何节点。注意，树中没有回路。跟随链接，最终会抵达“终点”。注意每个节点都有两个链接指向其他节点(或为 NULL 值)。这种形式的树称为二叉树，因为每个节点都恰好有两个链接。还有其他类型的树，其节点有不同的链接数量，但二叉树是所有树中最常见的。

树不是链表，但确实采用和链表相似的方法使用链接(指针)。在二叉树中，节点类型的定义在本质上等同于双链表的节点类型定义。但两个链接在命名时，通常采用 left 和 right 等单词(而不是 forward 和 back)。下面展示了用于构造二叉树的节点类型：

```
struct TreeNode
{
    int data;
    TreeNode *leftLink;
    TreeNode *rightLink;
};
```

图 13.12　二叉树

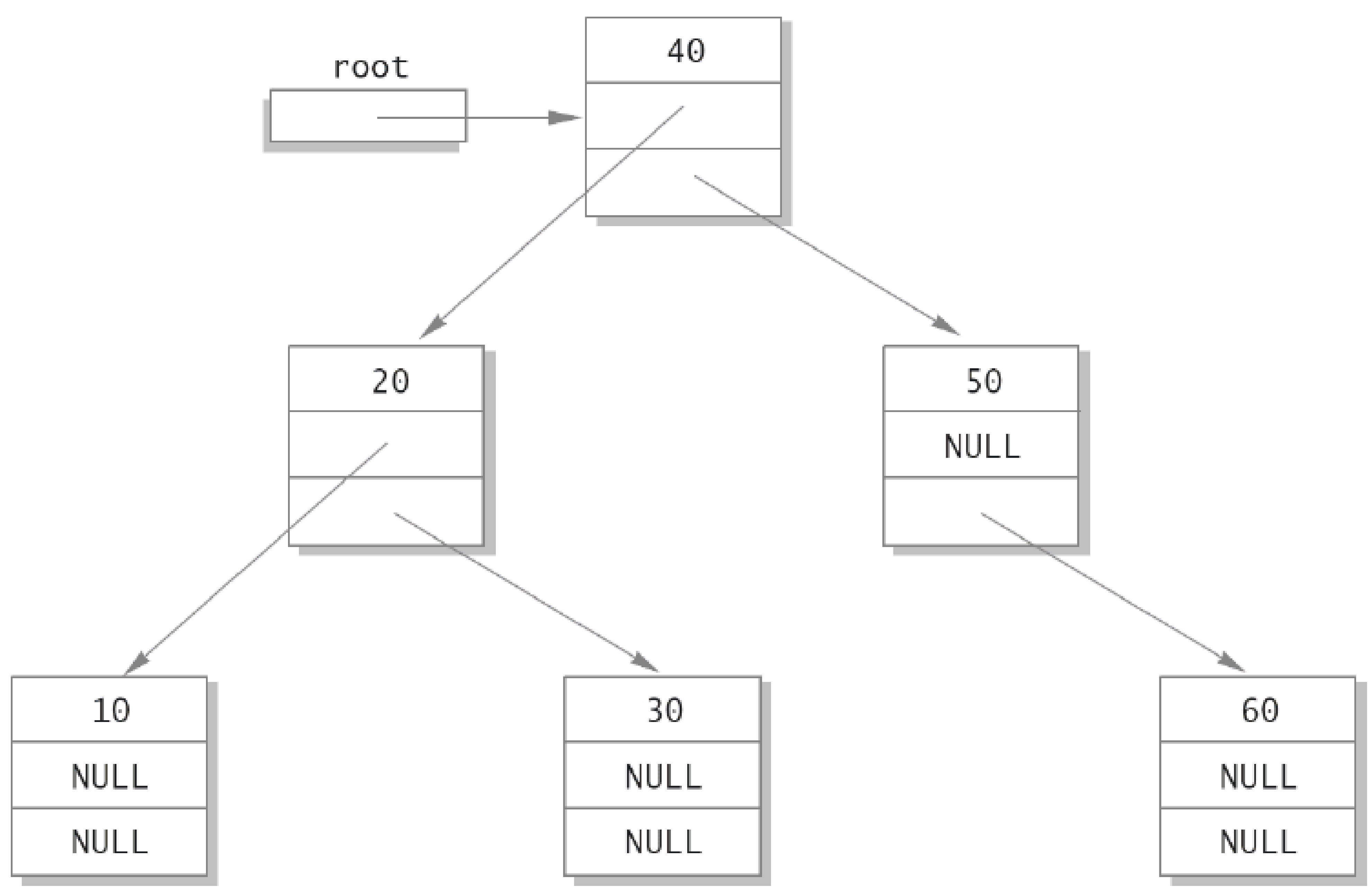


图 13.12 中，名为 root 的指针指向根节点(顶部节点)。**根节点**作用类似于普通链表(图 13.10)的表头节点。跟随链接，可从根节点抵达树中任何节点。

“树”这个术语或许不恰当。根跑到了树的顶部，分支看上去更像是根的分支，而不像是树的分支。事实上，这个术语要反映真实情况，必须将图 13.12 倒过来看。这样一来，该图就确实能反映树的分支，根节点就是树的根开始的地方。在分支结束处的节点中，两个链接变量都设为 NULL，这些节点称为**叶节点**(leaf node)。把图倒过来看，“叶节点”这个术语倒是挺贴切的。

虽然本书没有更深入讨论这方面的主题，但请记住，二叉树能高效存储和检索数据。

类构成的链表

视频讲解：*Walkthrough of Linked Lists of Classes*

前几个例子创建链表时，是用 struct 容纳链表节点内容。同样的数据结构还可用类代替 struct。基本原理一样，将 struct 的使用和定义语法替换成类的就可以了。

图 13.13 和图 13.14 演示如何定义 Node 类。根据信息隐藏原则，数据变量声明为 private，同时创建了 public 方法来访问数据值和链表的下一个节点。图 13.15 创建了一个只有 5 个节点的小链表，这是通过在链表前端插入新节点来完成的。headInsert 在逻辑上等价于图 13.4 定义的同名函数，只是它现在用 Node 类的构造函数设置数据。

图 13.13　Node 类的接口文件

```
1  // 头文件 Node.h
2  // 这是一个节点类的接口，该类与图 13.4 定义的结构具有相似的行为
3  namespace linkedlistofclasses
4  {
5    class Node
6    {
7    public:
8        Node();
9        Node(int value, Node *next);
```

```
10          // 初始化节点的构造函数
11
12          int getData() const;
13          // 检索这个节点的值
14
15          Node *getLink() const;
16         // 检索链表的下一个节点
17
18         void setData(int value);
19         // 修改链表中存储的值
20
21         void setLink(Node *next);
22         // 更改对下一个节点的引用
23
24     private:
25          int data;
26          Node *link;
27    };
28    typedef Node* NodePtr;
29  } // 类构成的链表
30  // Node.h
```

图 13.14　Node 类的实现文件

```
1  // 实现文件 Node.cpp
2  // 它实现了节点类的逻辑，接口见 Node.h
3  #include <iostream>
4  #include "Node.h"
5
6  namespace linkedlistofclasses
7  {
8      Node::Node() : data(0), link(NULL)
9      {
10          // 有意留空
11     }
12
13     Node::Node(int value, Node *next) : data(value), link(next)
14     {
15          // 有意留空
16     }
17
18          // 下面是取值和赋值函数
19
20     int Node::getData() const
21     {
22          return data;
23     }
24
26     Node* Node::getLink() const
26     {
27          return link;
28     }
29
30     void Node::setData(int value)
31     {
32          data = value;
33     }
34
35     void Node::setLink(Node *next)
36     {
37         link = next;
38     }
39  }  // 类构成的链表
40  // Node.cpp
```

图 13.15 使用 Node 类的程序

```
1  // 该程序演示了如何创建使用了 Node 类的链表
2  // 创建 5 个节点，输出它们，然后销毁
3  #include <iostream>
4  #include "Node.h"
5
6  using namespace std;
7  using namespace linkedlistofclasses;
8
9  // 该函数在链表表头插入新节点，
10 // 是图 13.4 同名函数的以类为基础的版本
11
12 void headInsert(NodePtr &head, int theNumber)
13 {
14    NodePtr tempPtr;
15    // 构造函数将 tempPtr->link 设为表头，
16    // 将数据值设为 theNumber
17    tempPtr = new Node(theNumber, head);
18    head = tempPtr;
19 }
20
21 int main()
22 {
23    NodePtr head, tmp;
24
25    // 创建以下节点的链表：4->3->2->1->0
26    head = new Node(0, NULL);
27    for (int i=1; i<5; i++)
28    {
29       headInsert(head, i);
30    }
31    // 遍历链表，显示每个值
32    tmp = head;
33    while (tmp != NULL)
34    {
35       cout << tmp->getData() << endl;
36       tmp = tmp->getLink();
37    }
38    // 退出程序前删除链表中的所有节点
39
40    tmp = head;
41    while (tmp != NULL)
42    {
43       NodePtr nodeToDelete = tmp;
44       tmp = tmp->getLink();
45       delete nodeToDelete;
46    }
47    return 0;
48 }
```

示范对话

```
4
3
2
1
0
```

13.2 栈 和 队 列

然而，有许多在先的，将要居后；许多居后的，将要在先。

——*马太福音 19 章 30 节*

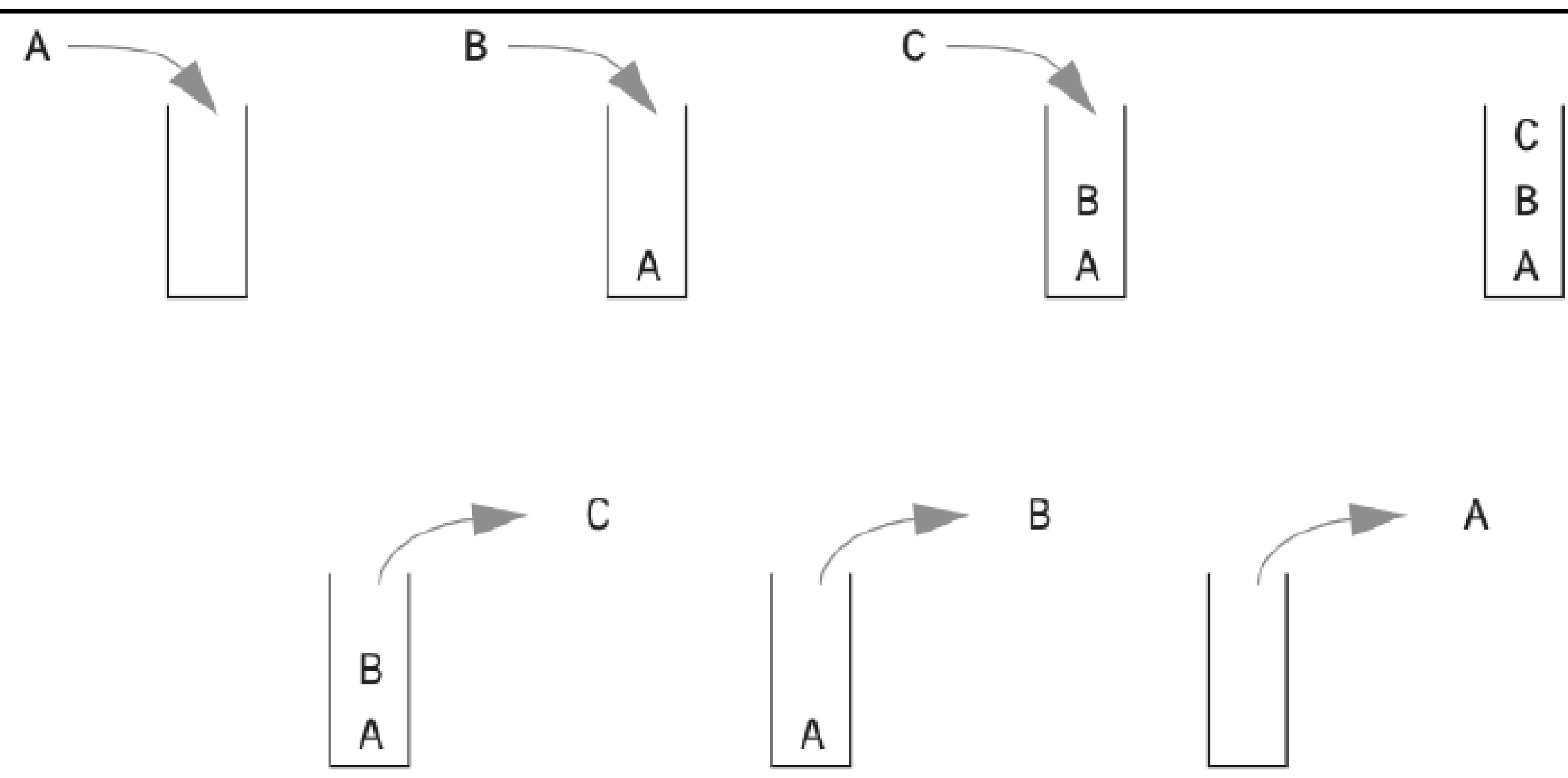
链表有多方面用途。本节展示它的两个应用实例。将用链表实现两个特殊的数据结构，一个是栈，另一个是队列。本节使用普通链表，不用双链表。

栈

栈是采取和存储数据时相反的顺序获取数据的一种数据结构。假定将字母'A'，'B'和'C'依次放入栈。从栈中取出时，首先取出'C'，然后是'B'，最后是'A'。图 13.16 展示了这一过程。如图 13.16 所示，可将栈想象成地上的一个洞。要从栈中取出一样东西，必须先删除它顶部的东西。考虑到这个原因，通常说栈是后入/先出(last-in/first-out，LIFO)数据结构。

栈有多种用途。第 14 章将讨论计算机系统如何用栈跟踪 C++函数调用。这里展示的是非常简单的应用程序。目标是展示如何用链表实现特定数据结构，栈是使用链表的简单例子。不用读第 14 章，就能理解这个例子。

图 13.16 栈



编程实例 栈类

栈类接口在图 13.17 给出。该栈用于存储 char 类型的数据。可定义类似的栈来存储其他任何类型的数据。可对栈执行两种基本操作：在栈中添加数据项，以及从栈中删除数据项。在栈中添加数据项，称为**压入**(push)或**入栈**，将相应成员函数命名为 push。从栈中删除

图 13.17 Stack 类的接口文件

```
1  // 头文件 stack.h：这是 Stack 类的接口，该类用于表示一个字符栈
2  #ifndef STACK_H
3  #define STACK_H
4
5  namespace stacksavitch
6  {
7      struct StackFrame
```

```
8       {
9           char data;
10          StackFrame *link;
11      };
12
13      typedef StackFrame* StackFramePtr;
14
15      class Stack
16      {
17      public:
18          Stack();
19          // 将对象初始化成空栈
20
21          Stack(const Stack& aStack);
22          // 拷贝构造函数
23
24          ~Stack();
25          // 销毁栈，将所有内存返还给自由存储
26
27          void push(char theSymbol);
28          // 后条件：theSymbol 添加到栈中
29
30          char pop();
31          // 前条件：栈非空
32          // 返回栈顶部的字符，并从栈删除那个字符
33
34          bool empty() const;
35          // 如果栈为空，就返回 true；否则返回 false
36      private:
37          StackFramePtr top;
38      };
39  } // stacksavitch
40
41  #endif // STACK_H
```

数据项，称为从栈中**弹出**(pop)或**出栈**，将相应成员函数命名为 pop。

push 和 pop 等名称源于栈的另一种比喻。这个比喻中，stack 就是餐厅用来收纳盘子的机构。该机构将所有盘子都存放在工作台上的一个洞中。盘子下方有一个弹簧，弹力进行了调整，确保只有最顶部的盘子才露出台面。将这种机构视为栈数据结构，就可在盘子上写数据(虽然有违食品健康法，但仍是一个很好的比喻)。要在栈中添加盘子，只需把它放在其他盘子顶部。新盘子的重量会压下弹簧。拿走一个盘子，下方的盘子会自动弹出以方便拿取。

图 13.18 是展示栈类用法的一个简单程序。该程序每次读取单词的一个字母，将字母放入栈中。然后，程序每次删除一个字母，把它们写到屏幕上。由于数据从栈删除时采取与入栈时相反的顺序，所以输出的单词是倒着写的。

图 13.18　使用 Stack 类的程序

```
1   // 该程序演示 Stack 类的使用
2   #include <iostream>
3   #include "stack.h"
4   using namespace std;
5   using namespace stacksavitch;
6
7   int main()
8   {
9       Stack s;
10      char next, ans;
11
12      do
13      {
```

```
14          cout << "Enter a word: ";
15          cin.get(next);
16          while (next != '\n')
17          {
18              s.push(next);
19              cin.get(next);
20          }
21
22          cout << "Written backward that is: ";
23          while ( ! s.empty() )
24              cout << s.pop();
25          cout << endl;
26
27          cout << "Again?(y/n): ";
28          cin >> ans;
29          cin.ignore(10000, '\n');
30      } while (ans != 'n' && ans != 'N');
31
32      return 0;
33  }
```

<第 8 章讨论了 cin 的 ignore 成员函数。本例除非遇到回车，否则会丢弃当前输入行剩余最多 10000 个字符的输入。行末的回车('\n')也会被丢弃>

示范对话

```
Enter a word: straw
Written backward that is: warts
Again?(y/n): y
Enter a word: C++
Written backward that is: ++C
Again?(y/n): n
```

如图 13.19 所示，栈类作为链表实现。其中，链表表头相当于栈顶。成员变量 top 是指向链表表头的指针。

图 13.19　Stack 类的实现

```
1  // Stack 类的实现文件 stack.cpp; Stack 类的接口位于头文件 stack.h 中
2  #include <iostream>
3  #include <cstddef>
4  #include "stack.h"
5  using namespace std;
6
7  namespace stacksavitch
8  {
9      // 使用 cstddef:
10     Stack::Stack() : top(NULL)
11     {
12         // 主体有意留空
13     }
14
15     Stack::Stack(const Stack& aStack)
16       <拷贝构造函数的定义见自测题 11>
17
18     Stack::~Stack()
19     {
20         char next;
21         while (! empty())
22            next = pop(); // pop 调用 delete
23     }
24
25     // 使用 cstddef:
26     bool Stack::empty() const
27     {
28         return (top == NULL);
29     }
30
31     void Stack::push(char theSymbol)
32       <剩余定义见自测题 10>
```

```
33
34     // 使用 iostream 和 cstdlib:
35     char Stack::pop()
36     {
37         if (empty())
38         {
39             cout << "Error: popping an empty stack.\n";
40             exit(1);
41         }
42
43         char result = top->data;
44
45         StackFramePtr tempPtr;
46         tempPtr = top;
47         top = top->link;
48
49         delete tempPtr;
50
51         return result;
52     }
53 } // stacksavitch
```

自测题 10 要求给出成员函数 push 的定义。但我们早已给出了这个任务的算法。成员函数 push 的代码与图 13.4 的 headInsert 函数基本相同，只是 push 要用一个名为 top 的指针代替 head 指针。

空栈就是空链表。所以要实现空栈，只需将 top 指针设为 NULL。知道 NULL 代表空栈后，如何实现默认构造函数和 empty 成员函数就一目了然了。

拷贝构造函数的定义复杂一些，但涉及的技术都是讨论过的。详情参见自测题 11。

pop 成员函数首先检查栈是否为空。非空就删除并返回栈顶的字符。首先设置局部变量 result，使之等于栈顶字符。如下所示：

```
char result = top->data;
```

将该字符保存到 result 变量后，top 指针就移至链表的下一个节点，这就相当于从列表删除了顶部节点。top 指针用以下语句来移动：

```
top = top->link;
```

但在 top 指针移动前，要先定位好名为 tempPtr 的临时指针，让它指向准备从列表中删除的节点。然后，调用 delete 销毁该节点：

```
delete tempPtr;
```

成员函数 pop 从链表删除的每个节点都调用 delete 销毁。所以，析构函数唯一要做的就是调用 pop 从栈中删除每个数据项。之后，每个节点占用的内存都会被自由存储回收。

√ 自测题

10. 给出图 13.17 描述的 Stack 类的成员函数 push 的定义。
11. 给出图 13.17 描述的 Stack 类的拷贝构造函数的定义。

队列

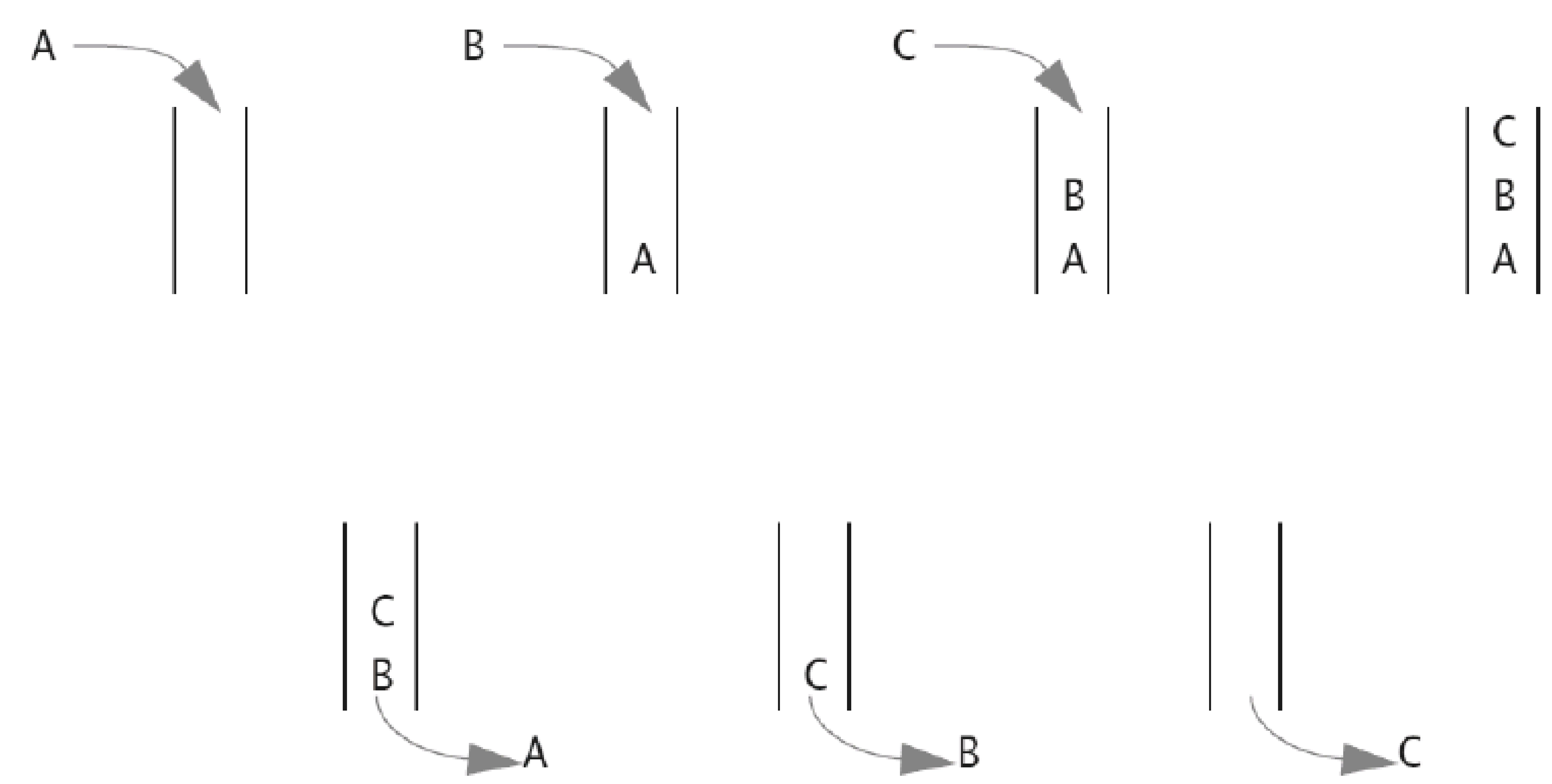
栈是后入/先出数据结构。另一种常见数据结构是**队列**(queue)，以**先入/先出**(first-in/first-out，FIFO)的方式处理数据。队列就像是在银行提款机或其他服务场所排队的一队人，按入队顺序提供服务。图 13.20 展示了一个队列的工作方式。

队　列

队列是先入/先出数据结构，数据项按加入队列的顺序从队列删除。

可采取与实现 Stack 类相似的方式，通过链表实现队列。但队列要求在链表头尾都安置一个指针，因为这两个地方都可能采取操作。从链表头部删除节点比从尾部删除更容易。所以我们的实现是从表头删除节点(现在称为列表的 front，或者队头)。另外，在列表另一端(现在称为列表的 back，或者队尾)添加节点。

图 13.20　队列



编程实例　队列类

图 13.21 展示了队列类的接口，该队列存储 char 类型的数据。可定义类似的队列来存储其他任意类型的数据。可为队列采取两种基本操作：在队尾添加数据项，以及从队头删除数据项。

图 13.22 的简单程序展示了队列类的用法。程序每次从单词中读取一个字母，并将字母置于队列。然后，程序一个接一个删除字母，将其写到屏幕上。由于数据按它们进入队列的顺序删除，所以输出的单词忠实反映了用户输入字母的顺序。最好将该队列应用程序与图 13.18 的栈应用程序对照一下。

图 13.21　Queue 类的接口文件

```
1  // 头文件 queue.h；这是 Queue 类的接口，该类用于表示一个字符队列
2  #ifndef QUEUE_H
3  #define QUEUE_H
4  namespace queuesavitch
5  {
6      struct QueueNode
7      {
8          char data;
9          QueueNode *link;
10     };
11     typedef QueueNote* QueueNotePtr;
12     class Queue
13     {
14      public:
15         Queue();
```

```
16          //将对象初始化成空队列
17          Queue(const Queue& aQueue);
18          ~Queue();

19          void add(char item);
20          //后条件：数据项已经添加到队列的末尾

21          char remove();
22          //前条件：队列为非空
23          //返回队列前端的数据项并从队列中删除该项

24          bool empty() const;
25          //如果队列为空，就返回 true；否则返回 false
26      private:
27          QueueNodePtr front;//指向链表表头，
28                            //数据项在表头删除
29          QueueNodePtr back; //指向链表另一端.数据项在这一端添加
30      };
31  }//queuesavitch
32  #endif //QUEUE_H
```

图 13.22 使用 Queue 类的程序

```
1  // 该程序演示 Queue 类的使用
2  #include<iostream>
3  #include"queue.h"
4  using namespace std;
5  using namespace queuesavitch;
6
7  int main()
8  {
9      Queue q;
10     char next, ans;
11
12     do
13     {
14         cout<<"Enter a word: ";
15         cin.get(next);
16         while(next != '\n');
17         {
18            q.add(next);
19            cin.get(next);
20         }
21
22         cout <<"You entered: ";
23         while ( ! q.empty() )
24            cout << q.remove();
25         cout << endl;
26
27         cout << "Again?(y/n): ";
28         cin >> ans;
29         cin.ignore(10000, '\n');
30      }while (ans != 'n' && ans != 'N');
31
32     return 0;
33 }
```

<第 8 章讨论了 cin 的 ignore 成员函数。本例除非遇到回车，否则会丢弃当前输入行剩余最多 10000 个字符的输入。行末的回车('\n')也会被丢弃>

示范对话

```
Enter a word: straw
You entered: straw
Again?(y/n): y
Enter a word: C++
You entered: C++
Again?(y/n): n
```

如图 13.21 和图 13.23 所示，队列类作为链表来实现。其中，表头是队列的 front。成员变量 front 是指向链表表头的指针。节点从链表表头删除。成员变量 back 也是指针，但它指向链表另一端的节点。节点在链表这一端添加。

空队列就是空链表。所以，为了实现空队列，只需将 front 和 back 设为 NULL。其他实现细节与上个例子相似，在此不再赘述。

图 13.23　Queue 类的实现

```
1  // 实现文件 queue.cpp
2  // 这是 Queue 类的实现
3  // Queue 类的接口在头文件 queue.h 中

4  #include <iostream>
5  #include <cstdlib>
6  #include <cstddef>
7  #include "queue.h"
8  using namespace std;
9
10 namespace QueueSavitch
11 {
12
13     //使用 cstddef:
14     Queue::Queue() : front(NULL), back(NULL)
15     {
16         //有意为空
17     }
18     Queue::Queue(const Queue& aQueue)
19          <拷贝构造函数的定义见自测题 12>
20     Queue::~Queue()
21          <析构函数的定义见自测题 13>
22
23     //使用 cstddef:
24     bool Queue::empty() const
25     {
26         return (back == NULL); // 可换成 front == NULL
27     }
28     //使用 cstddef:
29     void Queue::add(char item)
30     {
31         if(empty())
32         {
33            front = new QueueNode;
34            front->data = item;
35            front->link = NULL;
36            back = front;
37         }
38         else
39         {
40            QueueNodePtr tempPtr;
41            tempPtr = new QueueNode;
42            tempPtr->data = item;
43            tempPtr->link = NULL;
44            back->link = tempPtr
45            back = tempPtr;
46         }
47 }
48
49 //使用 cstdlib 和 iostream
50 char Queue::remove()
51 {
52     if (empty())
53     {
54         cout << "Error: Removing an item from an empty queue.\n";
55         exit(1);
56     }
57
```

```
58      char result = front->data;
59      QueueNodePtr discard;
60      discard = front;
61      front = front->link;
62      if (front == NULL) //如果已经删除了最后一个节点
62          back = NULL;
64      delete discard;
65
66      return result;
67      }
68  }//QueueSavitch
```

√ 自测题

12. 给出图 13.21 描述的 Queue 类的拷贝构造函数的定义。

13. 给出图 13.21 描述的 Queue 类的析构函数的定义。

小　　结

- 节点是一个 struct 或类对象，它的一个或多个成员变量是指针变量。这些节点通过指针成员变量连接，最终生成一个可在程序运行时自由伸缩的数据结构。
- 链表是节点构成的列表，每个节点都包含到下一个节点的指针。
- 要标记链表(或其他链接数据结构)的尾部，将指针成员变量设为 NULL 即可。
- 栈是可用链表实现的后入/先出数据结构。
- 队列是可用链表实现的先入/先出数据结构。

自测题答案

1.
```
Sally
18
18
18
```

注意： (*head).name 和 head->name 含义相同。(*head).number 和 head->number 含义也相同。

2. 最佳答案如下：

```
head->next = NULL;
```

但以下答案也正确：

```
(*head).next = NULL;
```

3.
```
delete head;
```

4.
```
head->item = "Wilbur's brother Orville";
```

5.
```
struct NodeType
{
    char data;
    NodeType *link;
};

typedef NodeType* PointerType;
```

6. 用指针变量 NULL 表示空列表。

7.
```
p1 = p1->next;
```

8.
```
Pointer discard;
discard = p2->next;
// 现在，discard 指向待删节点
p2->next = discard->next;
```

如只是从链表删除节点，上述语句足够。但该节点不再使用，应调用 delete 销毁，如下所示：

```
delete discard;
```

9. 正确答案是 a。在大型链表已知位置插入新数据项时，效率要高于在大型数组中插入。在链表中插入时，总共要执行大约 5 个操作，其中大多数都是指针赋值，与链表长度无关。相反，如果在数组中插入，平均要移动一半的数组元素，才能完成一个数据项的插入。

对于小型链表，正确答案应是 c。

10.
```
// 使用 cstddef：

void Stack::push(char theSymbol)
{
    StackFramePtr tempPtr;
    tempPtr = new StackFrame;

    tempPtr->data = theSymbol;

    tempPtr->link = top;
    top = tempPtr;
}
```

11.
```
// 使用 cstddef：
Stack::Stack(const Stack& aStack)
{
    if (aStack.top == NULL)
```

```
        top = NULL;
    else
    {
        StackFramePtr temp = aStack.top; // temp 在 aStack 中从头到尾遍历所有节点
        StackFramePtr end; // 指向新栈的尾部
        end = new StackFrame;
        end->data = temp->data;
        top = end;
        // 第一个节点已经创建，而且填充了数据
        // 现在，新节点要在这个节点之后添加

        temp = temp->link;
        while (temp != NULL)
        {
            end->link = new StackFrame;
            end = end->link;
            end->data = temp->data;
            temp = temp->link;
        }
        end->link = NULL;
    }
}
```

12.
```
// 使用 cstddef:
Queue::Queue(const Queue& aQueue)
{
    if (aQueue.empty())
        front = back = NULL;
    else
    {
        QueueNodePtr tempPtrOld = aQueue.front;
        // tempPtrOld 在 aQueue 中从 front 到 back，遍历所有节点
        QueueNodePtr tempPtrNew;
       // tempPtrNew 用于创建新节点

        back = new QueueNode;
        back->data = tempPtrOld->data;
        back->link = NULL;
        front = back;
        // 第一个节点已经创建，而且填充了数据
        // 现在，新节点要在这个节点之后添加

        tempPtrOld = tempPtrOld->link;
        // tempPtrOld 现在指向第二个节点；如果没有第二个节点，则为 NULL

        while (tempPtrOld != NULL)
        {
            tempPtrNew = new QueueNode;
            tempPtrNew->data = tempPtrOld->data;
            tempPtrNew->link = NULL;
            back->link = tempPtrNew;
            back = tempPtrNew;
            tempPtrOld = tempPtrOld->link;
        }
    }
}
```

13.
```
Queue::~Queue()
{
    char next;
    while (! empty())
        next = remove();  // remove 调用 delete
}
```

编 程 练 习

编程练习一般只需写很小的程序，运用本章提到的编程概念。

1. 以下程序创建含 3 个名字的链表。

```
#include <iostream>
#include <string>
```

```
using namespace std;

struct Node
{
   string name;
   Node *link;
};

typedef Node* NodePtr;

int main()
{
   NodePtr listPtr, tempPtr;
   listPtr = new Node;
   listPtr->name = "Emily";

   tempPtr = new Node;
   tempPtr->name = "James";
   listPtr->link = tempPtr;

   tempPtr->link = new Node;
   tempPtr = tempPtr->link;
   tempPtr->name = "Joules";
   tempPtr->link = NULL;

   return 0;
}
```

在 main 函数中添加代码做下面这些事情。

a. 顺序输出列表中的所有名字。

b. 在名字“James”后插入“Joshua”。输出修改后的列表。

c. 删除“Joules”节点。输出修改后的列表。

d. 删除列表中的所有节点。

2. 重做编程练习 1，这次使用 Node 类而不是结构。类要提供恰当的成员函数来设置名字和指向下一个节点的链接。还可考虑添加构造函数来设置名字和链接。

编 程 项 目

编程项目要求综合运用多方面的知识来解决问题，程序一般比编程练习大，解题方式多样化。

1. 写 void 函数获取整数链表，反转节点顺序。函数要接收一个传引用参数，它是指向表头的指针。调用函数后，该指针同样指向一个链表的表头。该链表包含的节点与原始链表相同，只是顺序相反。注意函数不能创建或销毁任何节点，只是重新排列节点。将函数放到合适的测试程序中。

2. 写 mergeLists 函数来获取两个传引用参数。两个参数都是指针变量，分别指向两个链表的表头，而且两个链表都由 int 类型的值构成。假定两个链表已排序，位于表头的数字是最小的数字，然后从小到大依次排列。函数返回指向一个新链表的表头的指针。新链表包含原来两个链表的所有节点，而且同样按从小到大的顺序排列。注意函数不能创建或销毁任何节点。函数调用结束后，两个指针变量参数的值应该为 NULL。

3. 设计和实现多项式类。以下多项式要作为链表实现：

$$a_nx^n + a_{n-1}x^{n-1} + \ldots + a_0$$

每个节点都包含一个代表 x 的乘方的 int 值(x^n)，以及一个代表相应系数的 int 值(a_n)。类支持的操作包括加法、减法、乘法和多项式求值。重载操作符+，-和*，以便直接用它们执行多项式的加法、减法和乘法运算。

多项式求值作为成员函数实现，它接收 int 类型的一个参数。求值成员函数用参数值取代 x，执行指定的运算，返回结果值。提供 4 个构造函数：一个默认构造函数；一个拷贝构造函数；一个接收单个

int 参数的构造函数，生成只有一个常量项的多项式，该常量就是构造函数的参数值；以及一个接收两个 int 参数的构造函数，生成只有一个项的多项式，它的系数和指数(例如 n)由两个参数指定。换言之，单参数构造函数生成的多项式实际是一种只由 a_0 构成的简单形式，双参数构造函数生成的多项式稍微复杂一些，代表 $a_n x^n$。提供恰当的析构函数。提供相应的成员函数来输入和输出多项式。

用户输入一个多项式时，可输入：

$$a_n x\text{^}n + a_{n-1} x\text{^}n-1 + \cdots + a_0$$

但是如果系数 a_i 等于 0，用户可省略 $a_i x\text{^}i$ 这一项。例如以下多项式：

$$3x^4 + 7x^2 + 5$$

可采取以下形式输入：

3x^4 + 7x^2 + 5

也可以采取以下形式输入：

3x^4 + 0x^3 + 7x^2 + 0x^1 + 5

如果系数为负，可用减号取代加号，例如：

3x^5 − 7x^3 + 2x^1 − 8

−7x^4 + 5x^2 + 9

多项式前的减号(例如第二个例子)只作用于第一个系数，而不是对整个多项式求反。多项式以相同的格式输出。输出时，系数为 0 的项不需要输出。

为简化输入，可假定多项式肯定是每行输入一个，而且肯定有一个常量项 a_0。如果没有常量项，用户要自己输入为 0 的常量项，如下所示：

12x^8 + 3x^2 + 0

4. 本项目要求用链表(而不是数组)重做第 11 章的编程项目 8。由于要生成由 double 数据项构成的链表，所以要对部分成员函数进行修改。成员包括：一个默认构造函数；一个名为 addItem 的成员函数，它在链表中添加一个 double 值；一个 full()函数，它检测列表是否已满，返回一个布尔值；以及一个友元函数，它重载插入操作符<<。

5. 编程项目 4 的一个更难的版本是写名为 List 的类。它与编程项目 4 相似，但包含一组非常全面的成员函数。

   - 默认构造函数 List()
   - ***double*** List::front();：返回列表第一项。
   - ***double*** List::back();：返回列表最后一项。
   - ***double*** List::current();：返回当前项。
   - ***void*** List::advance();：使 current()返回的项向前跳进一个。
   - ***void*** List::reset();：重置列表，使 current()返回列表第一项。
   - ***void*** List::insert(***double*** afterMe, ***double*** insertMe);：在 afterMe 之后插入 insertMe，递增私有变量 count。
   - ***int*** size();：返回列表中的数量项数量。
   - ***friend*** istream& ***operator***<< (istream& ins, ***double*** writeMe);

   私有数据成员如下：

```
node* head;
node* current;
int count;
```

   可能还要多加一个指针。需要为链表节点使用以下 struct(放在 List 类的外部)：

```
struct node
{
    double item;
    node *next;
};
```

增量开发对任何规模的项目都至关重要，本项目也不例外。先写 List 类的定义，暂不实现任何成员。将类定义放到文件 list.h。然后，在包含 int main(){}的文件中使用#include 包含"list.h"。编译文件。这样就能发现语法和打字错误。不进行这样的检查，以后实现成员时，可能产生许多不明不白的错误。检查通过后，每次只实现和编译一个成员，直至最终能在 main 函数中编写测试代码。

6. 视频讲解：*Solution to Programming Project 13.6*

   古代有一位美丽的公主 Eva，前来求婚的人络绎不绝。她决定按照下面的程序确定自己的意中人。首先，所有求婚者一个接一个地站成一列，按顺序领取号码。第一个求婚者是 1 号，第二个求婚者是 2 号，依此类推，第 *n* 个求婚者是 *n* 号。首先从第一个求婚者开始，她从队列中数出三个求婚者(因为她的名字 Eve 由三个字母组成)，第三个求婚者将被排除并出列。然后，Eve 接着往下数，再数出三个求婚者，将第三个求婚者排除在外。依此类推。到达队尾时，继续从头数。

   例如，假定有 6 个求婚者，排除过程如下所示：

   123456　最初的求婚者列表，从 1 开始数
   12456　排除第 3 个求婚者，从 4 开始数
   1245　排除第 6 个求婚者，从 1 开始数
   125　排除第 4 个求婚者，从 5 开始数
   15　排除第 2 个求婚者，从 5 开始数
   1　排除第 5 个求婚者，1 胜出

   写程序创建循环链表，演示当有 *n* 个求婚者时，你应该站在哪个位置才能让公主选中你。循环链表的特点是最后一个节点的链接字段引用第一个节点。为模拟筛选过程，程序要删除此过程中每一步被排除的求婚者。要考虑删除表头节点的可能性。

7. 重做(或第一次做)第 9 章的编程项目 5。不过，这次不用动态数组存储每一台计算机的用户 ID 列表，而是用一个链表。链表的节点包含计算机编号和登录到此计算机的人的用户 ID。如果没有人登录到此工作站，链表中就不应该有与此计算机对应的条目。

8. 修改或重写 Queue 类(图 13.21 到图 13.23)，模拟顾客去机动车管理局(Department of Motor Vehicles，DMV)排队办事的过程。顾客到那里会领到一个号。号从 1 开始，每个新顾客都+1。客服有空时，会按顺序叫号。因此，这个系统产生了一个 FIFO(先入先出)顾客队列，每个顾客都按号排序。写程序实现队列，模拟顾客进入和离开队列。队列由每个顾客拿到的号(ticket)构成。每个号的数据包括：号码(ticket number)和开始排号时的时间戳(time stamp)。客服开始接待一个顾客后，那个顾客的号码及其时间戳会被删除。程序应保存上三名顾客的排队时间。每次从队列中删除一个号(ticket)，都更新这些时间，输出上三名顾客的平均排队时间，将其作为下一名顾客被叫到号的估计时间。队列中没人了，就报告队列为空。

   下面给出了根据计算机时钟来计算时间戳的代码。在 C++的大多数实现中，time(NULL)函数都返回自 1970 年 1 月 1 日以来的秒数。

```
#include <ctime>
…
int main()
{
  long seconds;
  seconds = static_cast<long>(time(NULL));
  cout << "Seconds since 1/1/1970: " << seconds << endl;
  return 0;
}
```

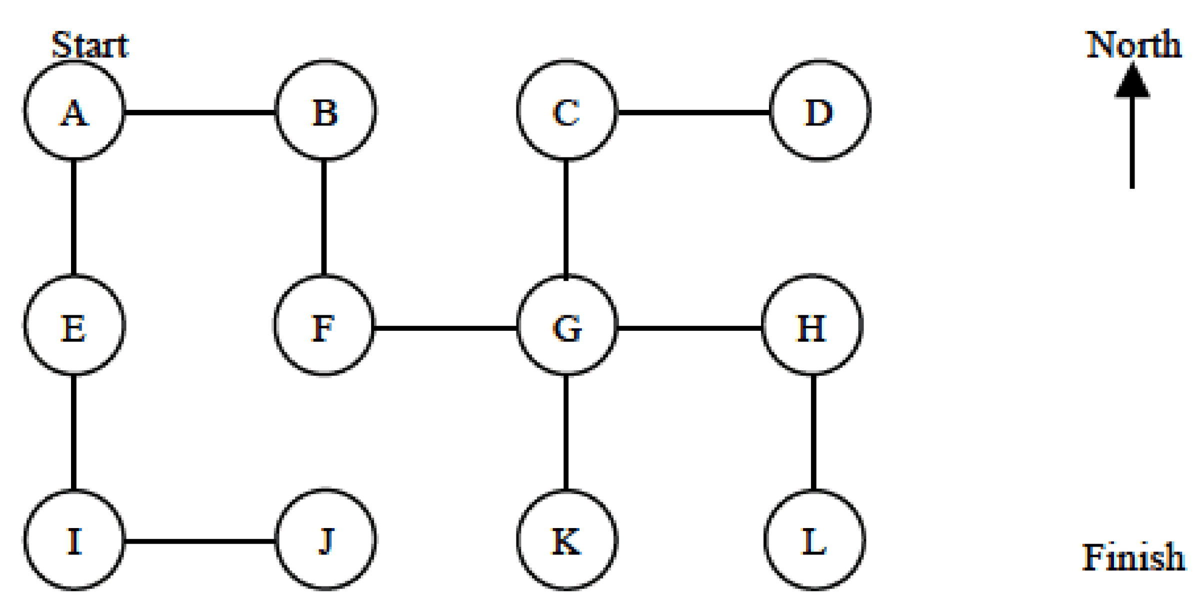
   下面是程序的一次示范执行情况：

```
The line is empty.
Enter '1' to simulate a customer's arrival, '2' to help the next customer, or '3' to quit..
1
Customer 1 entered the queue at time 100000044.
Enter '1' to simulate a customer's arrival, '2' to help the next customer, or '3' to quit..
1
Customer 2 entered the queue at time 100000049.
```

```
Enter '1' to simulate a customer's arrival, '2' to help the next customer, or '3' to quit..
1
Customer 3 entered the queue at time 100000055.
Enter '1' to simulate a customer's arrival, '2' to help the next customer, or '3' to quit..
2
Customer 1 is being helped at time 100000069. Wait time = 25 seconds.
The estimated wait time for customer 2 is 25 seconds.
Enter '1' to simulate a customer's arrival, '2' to help the next customer, or '3' to quit..
2
Customer 2 is being helped at time 100000076. Wait time = 27 seconds.
The estimated wait time for customer 3 is 26 seconds.
Enter '1' to simulate a customer's arrival, '2' to help the next customer, or '3' to quit..
1
Customer 4 entered the queue at time 100000080.
Enter '1' to simulate a customer's arrival, '2' to help the next customer, or '3' to quit..
2
Customer 3 is being helped at time 100000099. Wait time = 44 seconds.
The estimated wait time for customer 4 is 32 seconds.
```

9. 视频讲解：*Solution to Programming Project 13.9*

   下面描绘一个由圆和线构成的迷宫图。每个圆都是一个节点，可将这些圆想象成房间。一条线连接两个节点。注意，每个节点最多连接 4 条线。



   写程序用节点和指针实现该迷宫。节点用类或 struct 实现。线是双向链接，从一个节点指向另一个。迷宫起点是 A。用户目标是抵达节点 L 处的终点。程序应输出东南西北的所有可能的移动方向。程序的一次示范执行情况下如下：

```
You are in room A of a maze of twisty little passages, all alike.
You can go (E)ast, (S)outh, or (Q)uit.
E
You are in room B of a maze of twisty little passages, all alike.
You can go (W)est, (S)outh, or (Q)uit.
S
You are in room F of a maze of twisty little passages, all alike.
You can go (W)est, (S)outh, or (Q)uit.
E
```

10. 逆波兰表示法(Reverse Polish Notation，RPN)也称为后缀表示法，是指定数学表达式的一种格式。在 RPN 中，所有操作符都位于操作数后面；而在一般格式中，操作数位于操作数之间(所谓的中缀表示法)。从一个空栈开始，可用以下规则实现 RPN 计算器：

    - 输入数字，入栈
    - 输入"+"，最后两个操作数出栈，相加，结果入栈
    - 输入"-"，value1 出栈，value2 出栈，value2-value1 入栈
    - 输入"*"，最后两个操作数出栈，相乘，结果入栈
    - 输入"/"，value1 出栈，value2 出栈，value2/value1 入栈
    - 输入"q"，停止输入值，打印栈顶，退出程序

    修改 13.2 节的 Stack 类，存储整数而不是字符。使用修改的栈实现 RPN 计算器。给定一个操作符，但栈上没有两个操作数，就输出相应的错误消息。以下示例输入和输出对应于 ((10 - (2 + 3)) * 2)/5：

```
10
2
3
```

```
+
-
2
*
5
/
q
The top of the stack is: 2
```

11. 要求先完成编程项目 10。写程序将完全加好括弧的中缀表达式转换成等价的后缀表达式，并对后缀表达式求值。完全加好括弧的表达式是指为每个操作符及其操作数加上括弧。程序最开始有一个用于存储操作符的字符串空栈，还有一个用于存储后缀表达式的空字符串队列。转换规则如下：

- 输入"("，忽略
- 输入数字，入队
- 输入操作符("*"，"+"，"-"或"/")，入栈
- 输入")"，操作符出栈，入队
- 输入"q"，退出程序

最后一个操作符出栈时，队列中就包含了等价的后缀表达式。用编程项目 10 的解决方案求值这个表达式。要将字符串对象转换成整数。用 `c_str()`函数将字符串转换成 C 字符串，再用 `atoi` 函数将 C 字符串转换成整数。详情参考第 8 章。

以下是`((10 - (2 + 3)) * 2)`的示例输出，它转换成后缀表达式`10 2 3 + - 2 *`：

```
(
(
10
-
(
2
+
3
)
)
*
2
)
q
The expression evaluates to 10
```

第 14 章　递归

在一场关于宇宙学和太阳系结构的演讲后，威廉·詹姆斯被一个小个子老太太拦住了。

“你的理论说太阳是太阳系的中心，地球围着它转，詹姆斯先生，听起来蛮有道理，但实际上是错的。我有更好的理论。”小个子老太太说。

“喔，您的理论是什么呢，女士？”詹姆斯礼貌地问道。

“我们都站在一只大乌龟的背上，乌龟背上是一层厚厚的土。”

不想大费周章，詹姆斯决定以子之矛，攻子之盾。

“如果您的理论是正确的，女士，”他问道，“这只乌龟又站在什么上面呢？”

“你果然名不虚传，詹姆斯先生，我就知道你会这样问。”小个子老太太回答道，“但我早就有答案了：第一只乌龟站在第二只更大的乌龟身上，第二只乌龟就在第一只乌龟的下面。”

“但第二只乌龟又站在什么上面呢？”詹姆斯耐心地追问。

小个子老太太显得有些得意：“这是没用的，詹姆斯先生，乌龟们会一只一只叠起来的。”

——*J. R. ROSS，《变量语法的限制》*

概述

前面已遇到过几种循环定义的情况，但都获得圆满解决。在循环定义的各种例子中，最引人注目的就是一些 C++语句的定义。例如，while 语句的定义指出，它可以包含其他(较小的)语句。由于这些较小的语句可能是另一个 while 语句，所以造成了某种循环定义。将 while 语句的定义完整地写出来，其中就包含了对其他 while 语句的引用。在数学领域，这种循环定义称为递归定义。在 C++中，函数可采取相同的方式自己定义自己。准确地说，函数定义可包含对它自己的调用。在这种情况下，我们说这种函数是递归的。本章要详细讨论 C++的递归，并将递归当作一种编程和问题求解技术，从更广义的角度讨论它。

预备知识

14.1 节和 14.2 节的内容基于第 2 章～第 5 章的知识。14.3 节基于第 2 章～第 7 章以及第 10 章的知识。

14.1　面向任务的递归函数

我还想起一千零一夜正中间的那一夜，山鲁佐德王后(由于抄写员神秘的疏忽)开始一字不差地叙述一千零一夜的故事，这一来有可能又回到她讲述的那一夜，从而变得无休无止。

——*豪尔赫·路易斯·博尔赫斯[1](1899—1986)，《小径交叉的花园》*

---

① 豪尔赫·路易斯·博尔赫斯(Jorge Luis Borges)，阿根廷诗人、小说家兼翻译家。生于布宜诺斯艾利斯一个有英国血统的律师家庭。在日内瓦上中学，在剑桥读大学。掌握英、法、德等多国文字。中学时代开始写诗。1919 年赴西班牙，与极端主义派及先锋派作家过从甚密，共同编写文学期刊。1923 年出版第一部诗集，1935 年，小说集《恶棍列传》问世，引起阿根廷文学界重视。1941 年出版的小说集《小径交叉的花园》是他最具开拓意义的作品，此后《阿莱夫》《布罗迪报告》、《沙之书》和《莎士比亚的记忆》愈臻其妙，以丰富的叙事手法构筑了作者独有的迷宫式小说世界。——译注

写函数完成任务时，一种基本设计技术是将任务分解成子任务。有时，至少一个子任务是同一个任务的较小版本。例如，对于在数组中搜索特定值的任务，可把它分解成两个子任务：一个搜索数组前半部分，另一个搜索后半部分。对一半数组进行搜索的子任务是原始任务的“较小”版本。任何时候，只要子任务是原始任务的较小版本，就能用递归函数完成原始任务。需要经过一定的训练，才能习惯以这种方式分解问题。但在熟练之后，它就会成为设计算法的一种快捷方式，能帮助你快速设计出 C++函数。下面首先通过一个简单的案例分析来演示这种技术。

递　归

C++函数定义可包含对被定义函数自身的调用。这种情况下，就说函数是递归的。

案例分析：垂直数字

本案例要设计一个递归 void 函数，将一组数位垂直写到屏幕。例如，1984 会输出：

```
1
9
8
4
```

1. 问题定义

函数声明和注释如下：

```
void writeVertical(int n);
// 前条件：n >= 0
// 后条件：数字 n 垂直地写到屏幕上，每个数位单独占一行
```

2. 算法设计

有一种情况最简单。如果 n(要写到屏幕上的数字)是 1 位数，写这个数就行了。虽然简单，但这种情况非常重要，所以下面跟踪一下这种情况。

简单情况：如果 n < 10，就将数字 n 写到屏幕。

接着研究更普遍的情况，也就是数字由多个数位构成。假定要垂直地写 1234，那么屏幕输出如下：

```
1
2
3
4
```

为了将任务分解成两个子任务，可采用下面的方案。

方案 1：输出除最后一个数位之外的所有数位，比如：

```
1
2
3
```

方案 2：输出最后一个数位，本例是 4。

其中，子任务 1 是原始任务的较小版本，所以能用递归调用实现。子任务 2 是前面列

出的“简单情况”。所以，具有参数 n 的 writeVertical 函数的算法可以使用以下伪代码概括：

```
if (n < 10)
{
    cout << n << endl;
}
else // n 有 2 个或更多的数位：

{
    writeVertical(删除了最后一个数位的数字 n);   ←——— 递归的子任务
    cout << n 的最后一个数位 << endl;
}
```

转换成 C++代码时，只需将以下两句伪代码转换成 C++表达式：

删除了最后一个数位的数字 n
n 的最后一个数位

使用整数除法操作符/和取模操作符%，上述表达式可轻松转换成 C++表达式：

```
n/10      // 删除了最后一个数位的数字 n
n%10      // n 的最后一个数位
```

例如，1234/10 的求值结果是 123，而 1234%10 的求值结果是 4。

几个因素影响了两个子任务的选择。一个是能轻松计算对 writeVertical 函数进行递归调用时要传递的实参。“删除了最后一个数位的数字 n”可用 n/10 轻松求值获得。有人可能想这样分解子任务。

(1) 输出 n 的第一个数位。

(2) 输出删除了第一个数位的数字 n。

同样能将任务分解成子任务，而且同样能用递归方式实现。但很难计算删除第一个数位的结果，而计算删除最后一个数位的结果很容易。

之所以这样分解任务，另一个因素是其中一个子任务不涉及递归调用。在成功的递归函数定义中，至少要有一种情况不涉及递归调用(同时，一种或更多情况至少涉及一个递归调用)。下一节将详细讨论递归算法的这个问题。

3. 编码

综上所述，可以生成如图 14.1 所示的递归函数 writeVertical。下一节会更详细地解释本例的递归是如何进行的。

图 14.1　递归输出函数

```
1  // 该程序用于演示递归函数 writeVertical
2  #include <iostream>
3  using namespace std;
4
5  void writeVertical(int n);
6  // 前条件：n >= 0
7  // 后条件：数字 n 垂直地写到屏幕上，每个数位单独占一行
8
9  int main()
10
11     cout << "writeVertical(3):" << endl;
12     writeVertical(3);
```

```
13     cout << "writeVertical(12):" << endl;
14     writeVertical(12);
15
16     cout << "writeVertical(123):" << endl;
17     writeVertical(123);
18
19     return 0;
20 }
21
22 // 使用 iostream:
23 void writeVertical(int n)
24 {
25     if (n < 10)
26     {
27        cout << n << endl;
28     }
29     else // n 有 2 个或更多的数位:
30     {
31        writeVertical(n/10);
32        cout << (n%10) << endl;
33     }
34 }
```

示范对话

```
writeVertical(3):
3
writeVertical(12):
1
2
writeVertical(123):
1
2
3
```

4. 跟踪递归调用

现在分析进行以下函数调用时，具体会发生什么事情。

```
writeVertical(123);
```

处理该函数调用时，计算机的处理方式和处理其他任何函数无异。123 会替代函数定义中的参数 n，并执行函数体。用 123 替代 n 后，要执行的代码变成下面这样：

```
if (123 < 10)
{
    cout << 123 << endl;
}
else // n 有 2 个或更多的数位:
{
    writeVertical(123 / 10);   <——  计算将在此暂停，直到递归调用返回
    cout << (123 % 10) << endl;
}
```

由于 123 不小于 10，所以 if-else 语句中的逻辑表达式为 false，会执行 else 部分。然而，else 部分开始于以下函数调用：

```
writeVertical(n / 10);
```

它等价于以下调用(n 等于 123)：

```
writeVertical(123 / 10);
```

它等价于以下调用：

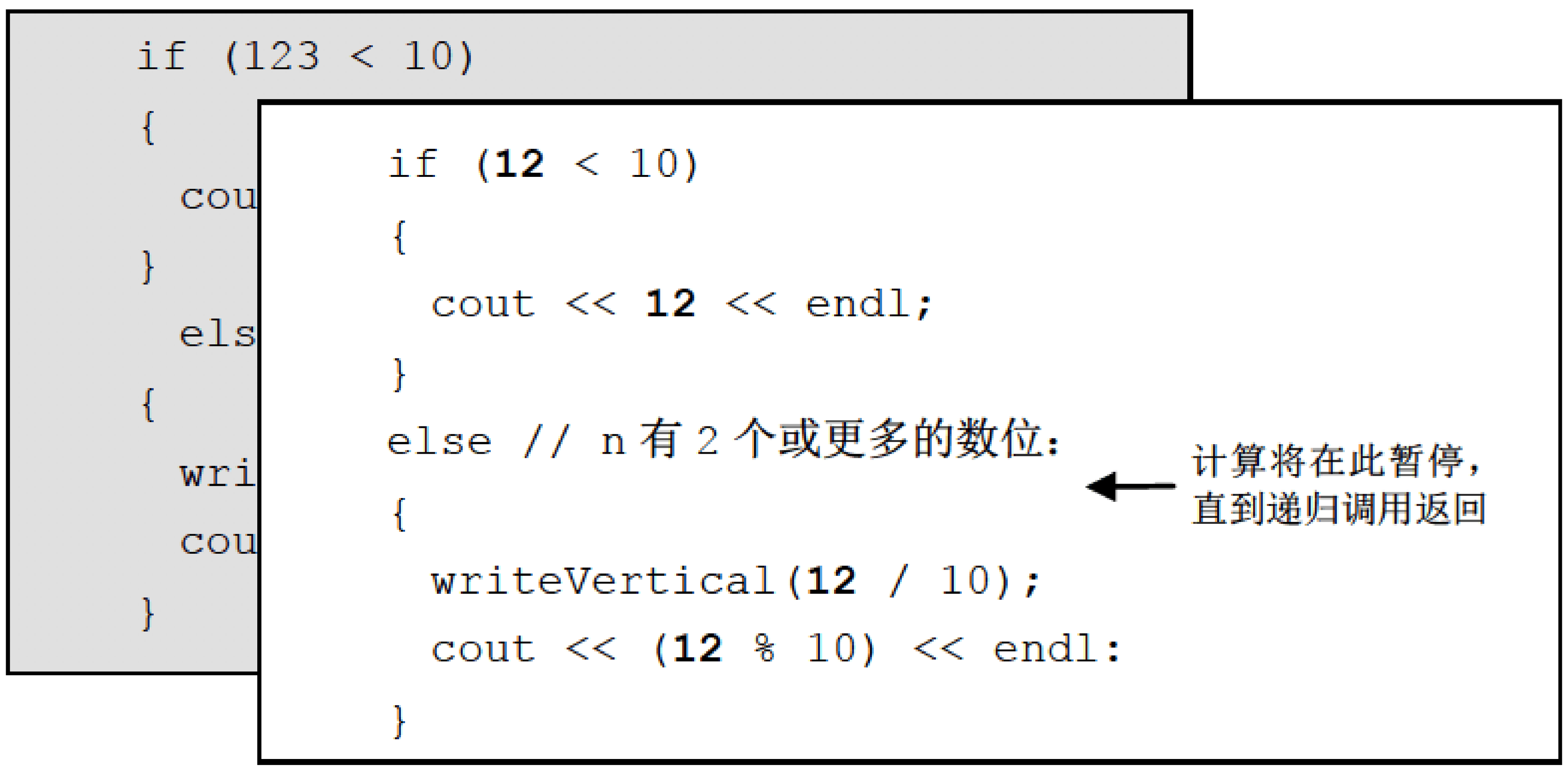
```
writeVertical(12);
```

一旦抵达这个递归调用，当前函数计算就会暂停，转而执行这个递归调用。递归调用结束后，会恢复被暂停的计算，并从这个位置继续。

以下递归调用：

```
writeVertical(12);
```

它的工作方式与其他任何函数调用相似。实参 12 会替代形参 n，再执行函数体。用 12 替代了 n 之后，就同时出现了两个函数计算，一个暂停，一个活动，如下所示。



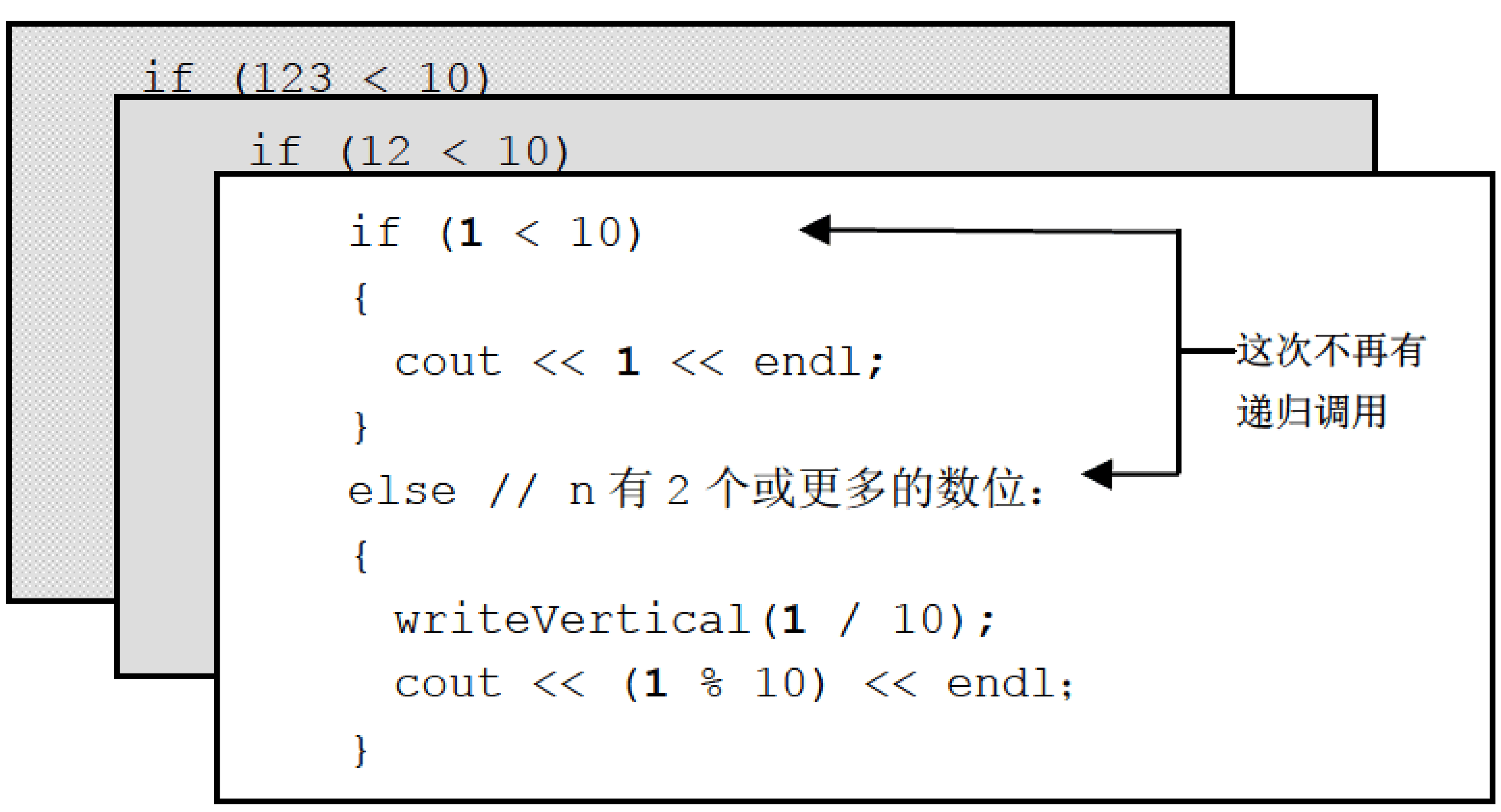
由于 12 不小于 10，所以 if-else 语句中的布尔表达式为 false，执行的将是 else 部分。但如前所述，else 部分以一个递归调用开始。递归调用的参数是 n/10，目前相当于 12/10。所以，writeVertical 的第二个计算也会暂停，并执行以下递归调用：

```
writeVertical(12 / 10);
```

它等价于以下形式：

```
writeVertical(1);
```

目前有两个暂停的计算等待恢复，计算机开始执行这个新的递归调用，它的工作方式与之前的递归调用相似。实参 1 会替代形参 n，再执行函数体。在这个时候，实际执行的计算如下所示：



这次执行函数体时，情况稍有不同。由于 1 小于 10，所以 if-else 语句中的表达式为 true，所以执行的是 else 之前的语句，也就是一个 cout 语句，它将参数值 1 写到屏幕。所以，writeVertical(1)调用的结果是将 1 写到屏幕后终止，不再有任何递归调用。

writeVertical(1)调用终止后，正在等待它终止的那个暂停的计算会恢复，如下所示：

```
if (123 < 10)
    if (12 < 10)
    {
      cout << 12 << endl;
    }
    else // n 有 2 个或更多的数位:
    {
      writeVertical(12 / 10);
      cout << (12 % 10) << endl: ◄──── 计算在此恢复
    }
```

这个暂停的计算恢复后，会执行一个 cout 语句，输出值 12%10，结果是 2。之后就会终止当前计算，但还有一个被暂停的计算正在等待恢复。恢复被暂停的最后一个计算时，情况如下所示：

```
if (123 < 10)
{
   cout << 123 << endl;
}
else //n is two or more digits long;
{
   wirite_vertical(123 / 10);
   count << (123 % 10) << endl; ◄──── 计算在此恢复
}
```

最后一个被暂停的计算恢复后，会输出值 123 % 10，结果是 3。至此，原始函数调用就可以终止了。结果是数位 1，2 和 3 都已依次写到屏幕上，每个一行。 ■

深入递归

writeVertical 函数的定义使用了递归。对函数调用 writeVertical(123)进行求值时，没有做任何新的或不同的事情。相反，和本书以前各章讲述的任何函数一样，用实参 123 替代形参 n，然后执行函数体的代码。抵达以下递归调用时：

```
writeVertical(123/10);
```

计算机采取以下方式跟踪递归调用。调用函数时，计算机在形参位置插入实参，开始执行

代码。如遇到递归调用，就暂停计算。这是由于它必须在知道了递归调用的结果之后才能继续。计算机保存以后继续计算所需的全部信息，再开始对递归调用进行求值。递归调用结束后，计算机会返回当初暂停的位置，继续外面一层的计算。

C++函数定义不限制递归调用的使用方式。但为了让递归函数定义真正有用，应确保任何函数调用最终都由不依赖递归的代码终止。函数可调用自身，递归调用可以再次调用函数。这个过程能重复任意多次。但除非最终有一个递归调用不依赖递归，否则这个过程无法终止。成功的递归函数定义可概括为以下两点。

- 在一种或多种情况下，函数通过递归调用完成任务的一个或多个较小版本。
- 在一种或多种情况下，函数不使用任何递归调用完成任务。这种不再产生递归调用的情况称为**基本情况**(base case)或**停止情况**(stopping case)。

通常由一个 `if-else` 语句判断具体执行哪种情况。初始函数调用执行的情况通常包含一个递归调用。该递归调用执行另一种要求递归调用的情况。递归调用可以不断地产生其他递归调用，但最终会遇到一种停止情况。**记住，每个函数调用最终都必须遇到停止情况，否则会因为递归调用无限继续下去，造成函数调用永远都不结束**(事实上，无限递归调用最后都会异常终止，而不是真的永远运行)。

为保证最终遇到停止情况，最常见的办法就是写一个函数，使某个(正)数在每次递归调用时都递减，一旦递减至某个“小值”，就表明遇到了停止情况。图 14.1 基于该思路来设计 `writeVertical` 函数。调用函数时会产生一个递归调用，并向该递归调用传递一个较小的值。每个递归调用都产生另一个递归调用，直至参数值小于 10，此时函数调用终止，不再产生更多递归调用。该过程逐渐倒退回原始调用并终止。

递归函数定义的常规形式

成功的递归函数定义具有以下常规形式。

- 在一种或多种情况下，对正在定义的函数执行一次或多次递归调用。这些递归调用被用来解决当前要解决之任务的更小版本。
- 在一种或多种情况下，不包括任何递归调用。这些无任何递归调用的情况称为基本情况或停止情况。

陷阱：无穷递归

在前面讨论的 `writeVertical` 函数中，递归调用最终会抵达一个不涉及任何递归的函数(也就是抵达一个停止情况)。如果结局不是这样的，而是每个递归调用都总是产生另一个递归调用，函数调用理论上会永远运行。这称为**无穷递归**(infinite recursion)。但在实际应用中，这样的函数会在计算机资源耗尽之后造成程序异常终止。总之，递归定义不应该是“一直递归”的。否则，就像本章开头老太太对地球的理解一样，函数调用永远都不结束，除非遇到异常情况。

无穷递归的例子不难找。以下是语法正确的 C++函数定义，可把它视为 `writeVertical` 的另一个版本：

```
void newWriteVertical(int n)
```

```
    {
        newWriteVertical(n / 10);
        cout << (n % 10) << endl;
    }
```

将上述定义嵌入一个程序，并在程序中调用该函数，编译器会成功编译，不会报错。另外，上述定义甚至貌似有一定道理。它表示为了输出 newWriteVertical 的参数值，首先要输出除最后一个数位之外的所有数位，再输出最后一个数位。但在调用时，该函数会生成一系列无穷尽的递归调用。如调用 newWriteVertical(12)，执行会暂停，转而执行递归调用 newWriteVertical(12/10)，它等价于 newWriteVertical(1)。然后又会暂停，产生一个新的递归调用，如下所示：

```
newWriteVertical(1/10);
```

它等价于以下形式：

```
newWriteVertical(0);
```

上述调用又会暂停，产生递归调用 newWriteVertical(0/10);，它也等价于以下形式：

```
newWriteVertical(0);
```

从此会不停执行同一个递归函数调用 newWriteVertical(0);。由于函数定义没有停止情况，所以该过程会一直继续（直至耗尽计算机资源）。■

√ 自测题

1. 以下程序输出什么？

```
#include <iostream>
using namespace std;
void cheers(int n);

int main()
{
    cheers(3);
    return 0;
}

void cheers(int n)
{
   if (n == 1)
    {
        cout << "Hurray\n";
    }
    else
    {
        cout << "Hip ";
        cheers(n - 1);
    }
}
```

2. 写递归的 void 函数接收一个正整数作为参数，在一行上输出恰好那么多个星号('*')。
3. 写递归的 void 函数接收一个正整数作为参数，在屏幕上逆序写出这个数字。例如，假定参数值是 1234，则输出 4321。
4. 写递归的 void 函数接收单个 int 参数 n，在屏幕上输出整数 1，2，…，n。
5. 写递归的 void 函数接收单个 int 参数 n，在屏幕上输出整数 n，n-1，…，3，2，1。**提示：**自测题 4 和自测题 5 的代码是共通的，只需修改两行代码即可。

用于递归的栈

为了跟踪递归和其他许多操作，大多数计算机会使用名为栈的结构。**栈**(stack)是一种特别的内存结构，可把它想象成一叠纸。要记一些东西，拿一张白纸，在上面写好字之后，把它放在那叠纸的顶部。要记录更多东西，可以再拿到一张白纸，同样将新纸放在那叠纸的顶部。采取这种方式，可以在栈(那叠纸)中放入越来越多的信息。

从栈中获取信息同样很简单。阅读最顶部的那张纸，看完后扔掉。但比较麻烦的是，只有最顶部那张纸才能使用。例如，要阅读上数第 3 张纸的内容，必须先把在它上面的 2 张纸扔掉。由于最后放到栈的纸最先离开栈，所以栈是**后入/先出**(last-in/first-out，LIFO)内存结构。

计算机使用栈可以轻松跟踪递归。任何时候只要调用一个函数，就拿一张白纸。函数定义复制到纸上，实参替换形参。然后，计算机执行函数体。一旦遇到递归调用，就暂停那张纸上的计算，转而执行递归调用。但执行递归调用前需保存足够多的信息，确保递归调用结束后能继续被暂停的计算。这些信息会记录到纸上，入栈。然后，再拿一张纸执行递归调用。计算机在纸上记录函数定义的第二个拷贝，实参替换形参，开始执行递归调用。遇到另外一个递归调用，就重复将信息保存到栈的过程，再拿一张纸执行新的递归调用。上一节跟踪递归调用时已图示了这一过程。虽然当时没把它叫做栈，但将一个计算叠加到另一个计算的上面，已清楚展示了栈的工作原理。

▶ 视频讲解：*Recursion and the Stack*

该过程会一直继续，直至某个递归调用在结束计算后，没有产生任何更多递归调用。届时，计算机会将注意力转向栈顶的那张纸。在这张纸中，包含了已部分完成的计算，它正在等待刚才结束的递归计算，以便从当初暂停的地方继续执行。执行完毕后，计算机会扔掉和这个计算对应的纸。因此，栈中位于它下方的计算变成栈顶，计算机将注意力转向目前位于栈顶尚未完成的计算……依此类推。该过程会持续到最底部的那张纸计算完毕，通常将这个过程称为**解链**或**辗转开解**。取决于产生了多少个递归调用，以及函数定义的编写方式，栈可以随意增大或缩小。注意在任何时候，栈中的纸只能采取后入/先出的方式使用，但那正是跟踪递归调用需要的方式。每个被暂停的版本都只有在它正上方的那个版本完成后才能继续。

当然，计算机并不是真的用纸记录信息，这只是比喻。计算机使用的是内存中的特定区域，而不是纸张。每个特定内存区域(一张纸)的内容都称为一个**活动帧**(activation frame)。这些活动帧采用刚才所说的后入/先出方式进行处理。注意，活动帧实际不包含函数定义的完整拷贝，而是全都引用同一个函数定义拷贝。不过，活动帧已包含了足够多的信息，允许每个活动帧都像包含了函数定义的一个完整拷贝那样工作。

栈

栈是后入/先出内存结构。从栈中引用或删除的第一个项肯定是最后一个入栈的项。计算机用栈跟踪递归(以及用于其他目的)。

陷阱：栈溢出

栈长度有限。如函数递归调用自己多次，每个递归调用都要求在栈中放入一个新的活动帧。调用链太长，可能造成栈长度超过限制，从而发生**栈溢出**(stack overflow)。如程序显示错误消息，报告 stack overflow，就可能是因为某个函数产生了太长的递归调用链。栈溢出的常见原因是无穷递归。函数无穷递归，再大的栈也不够用。■

递归与迭代

递归并非必不可少。事实上，有的编程语言根本不支持递归。能用递归完成的任务肯定能用其他方式完成。例如，图 14.2 展示的就是图 14.1 的函数的非递归版本。函数的非递归版本通常要用某种形式的循环(可能需要多个循环)代替递归。因此，一般将非递归版本称为**迭代版本**(iterative version)。将图 14.1 给出的 `writeVertical` 函数的定义替换成如图 14.2 所示的版本，输出结果完全一样。这个例子证明，某些情况下，函数的递归版本比迭代版本简单得多。

相较于迭代版本，递归版本通常运行得更慢，而且会使用更多存储空间。虽然从表面看 `writeVertical` 的迭代版本(图 14.2)比递归版本(图 14.1)使用的存储空间更多，执行的计算更多，但两个版本实际使用的存储空间是相当的，计算量也相当。事实上，递归版本反而可能使用更多的存储空间，而且运行得更慢，因为计算机必须花不少工夫去操纵栈以跟踪递归。但由于系统是自动完成这些工作，所以递归有时能简化程序员的工作，有时还能生成更容易理解的代码。正如本章的例子以及自测题/编程项目所展示的那样，某些情况下，递归定义显得更简单、更清晰；另一些情况下，迭代定义则显得更简单、更清晰。

图 14.2 图 14.1 的函数的迭代版本

```
1  // 使用 iostream:
2  void writeVertical(int n)
3  {
4      int tensInN = 1;
5      int leftEndPiece = n;
6      while (leftEndPiece > 9)
7      {
8          leftEndPiece = leftEndPiece/10;
9          tensInN = tensInN*10;
10     }
11     // tensInN 是 10 的乘方，具有与 n 同样多的数位
12     // 例如，假如 n=2345，则 tensInN=1000
13
14     for (int powerOf10 = tensInN;
15            powerOf10 > 0; powerOf10 = powerOf10/10)
16     {
17         cout << (n/powerOf10) << endl;
18         n = n%powerOf10;
19     }
20 }
```

√ 自测题

6. 如程序报错，说“栈溢出”或“堆栈溢出”(stack overflow)，错误根源可能在哪里？
7. 为自测题 1 定义的 cheers 函数写迭代版本。

8. 为自测题 2 定义的函数写迭代版本。
9. 为自测题 3 定义的函数写迭代版本。
10. 请跟踪一遍你为自测题 4 设计的递归方案。
11. 请跟踪一遍你为自测题 5 设计的递归方案。

14.2 面向值的递归函数

递归只应天上有，凡人该当用迭代。

——*无名氏*

要返回值的递归函数的常规形式

前面所见的递归函数都是 void 函数，但递归不限于 void 函数。递归函数可返回任何类型的值。设计递归函数返回值时，采用的技术与 void 函数基本相同。要返回值的一个成功的递归函数定义可概括如下。

- 在一种或多种情况下，通过调用相同的函数(也就是使用递归调用)来计算返回值。和 void 函数的情况一样，递归调用所用的参数值在直观上应该越来越“小”。
- 在一种或多种情况下，不用任何递归调用就能计算出返回值。没有递归调用的情况称为**基本情况**或**停止情况**(和 void 函数一样)。

下面的“编程实例”将具体演示这种技术。

编程实例 另一个求乘方函数

第 4 章介绍了用于求乘方的预定义函数 pow。例如，pow(2.0, 3.0)返回 $2.0^{3.0}$，所以以下语句设置变量 x，让它等于 8.0：

```
double x = pow(2.0, 3.0);
```

pow 函数获取 double 类型的两个实参，返回 double 类型的值。图 14.3 包含一个函数的递归定义，它能完成的工作与 pow 类似，只是使用 int 类型，而不是 double 类型。新函数叫 power。例如，以下语句设置 y 值，让它等于 8，因 2^3 等于 8：

```
int y = power(2, 3);
```

之所以要定义 power 函数，主要是我们想展示递归函数的一个简单例子。但有的时候，power 比 pow 更好用。pow 返回的 double 值只是约值。相反，power 函数返回 int 值，是准确值。某些时候，你可能正好需要 power 函数所赋予的额外精度。

图 14.3 递归函数 power

```
1  // 该程序用于演示递归函数 power
2  #include <iostream>
3  #include <cstdlib>
4  using namespace std;

5  int power(int x, int n);
```

```
 6  // 前条件: n >= 0
 7  // 返回 x 的 n 次方

 8  int main()
 9  {
10      for (int n = 0; n < 4; n++)
11           cout << "3 to the power " << n
12                << " is " << power(3, n) << endl;

13      return 0;
14  }
15  // 使用 iostream 和 cstdlib:
16  int power(int x, int n)
17  {
18      if (n < 0) // n 值不合法
19      {
20          cout << "Illegal argument to power.\n";
21          exit(1);
22      }

23      if (n > 0)
24          return ( power(x, n - 1)*x );
25      else // n == 0
26          return (1);
27  }
```

示范对话

```
3 to the power 0 is 1
3 to the power 1 is 3
3 to the power 2 is 9
3 to the power 3 is 27
```

power 函数的定义基于以下公式：

$$x^n = x^{n-1} * x$$

将公式转换为 C++代码，power(x, n)的返回值等于以下表达式的值：

```
power(x, n - 1) * x
```

图 14.3 的 power 函数定义正是为 power(x, n)返回这个值，前提是 n > 0。n 等于 0 的情况是停止情况。如 n 等于 0，则 power(x, n)返回 1(因 x^0 等于 1)。

下面用一些示范值调用 power 函数，看看具体会发生什么事情。首先分析下面这个简单的表达式：

```
power(2, 0)
```

调用函数时，x 的值设为 2，n 的值设为 0，执行函数体。由于 n 值合法，所以执行 if-else 语句。由于 n 的值不大于零，所以执行 else 之后的 return 语句，函数调用返回 1。因此，以下语句将 y 值设为 1：

```
int y = power(2, 0);
```

再来分析牵涉递归调用的例子。例如以下表达式：

```
power(2, 1)
```

调用函数时，x 设为 2，n 设为 1，执行函数体。由于 n 值大于零，所以用以下 return 语句确定返回值：

```
return ( power(x, n - 1) * x );
```

目前，它相当于以下形式：

```
return ( power(2, 0) * 2 );
```

至此，power(2, 1)的计算会暂停，这个暂停的计算的一个拷贝会放到栈中，计算机开始一个新的函数调用，计算 power(2, 0)的值。如前所述，power(2, 0)的值是 1。确定了 power(2, 0)的值之后，计算机将表达式 power(2, 0)替换成相应的值 1，并恢复当初暂停的计算。根据上述 return 语句，恢复的计算要像这样确定 power(2, 1)的终值：power(2, 0) * 2 相当于 1 * 2，结果是 2，power(2, 1)返回的终值是 2。因此，以下语句会将 z 的值设为 2：

```
int z = power(2, 1);
```

随着第二个参数值的逐渐增大，会产生越来越长的递归调用链。例如以下语句：

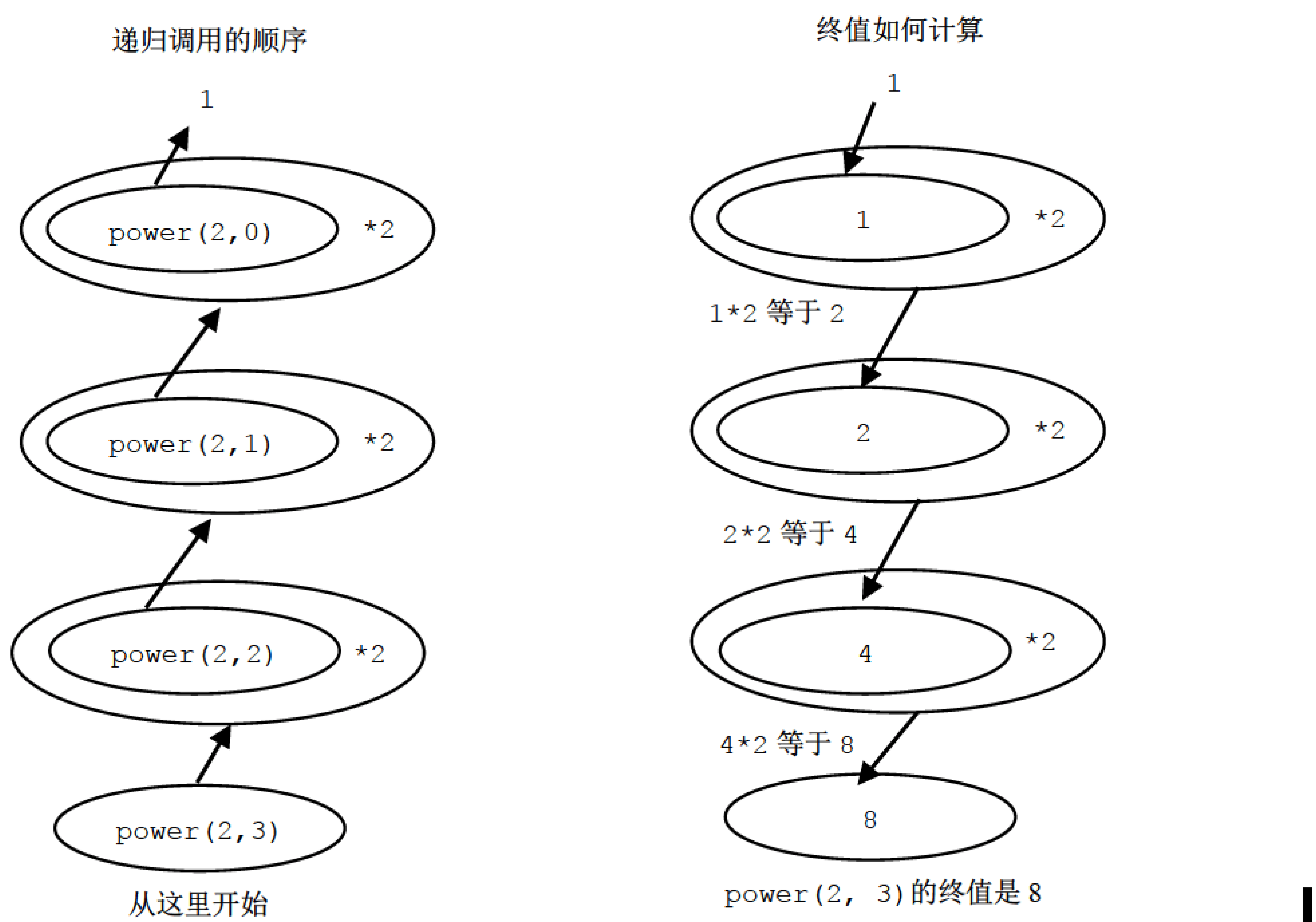
```
cout << power(2, 3);
```

power(2, 3)的值像下面这样计算：

```
power(2, 3)等于 power(2, 2) * 2
power(2, 2)等于 power(2, 1) * 2
power(2, 1)等于 power(2, 0) * 2
power(2, 0)等于 1(停止情况)
```

计算机抵达停止情况时，也就是遇到 power(2, 0)时，已产生了三个被暂停的计算。计算好停止情况的返回值之后，它恢复最近一次暂停的计算，确定 power(2, 1)的值。之后，计算机完成剩余每一个被暂停的计算，每次都将计算好的值插入另一个暂停的计算中，直至抵达并完成原始调用的计算，也就是 power(2, 3)。图 14.4 详细展示了整个计算过程。

图 14.4　递归函数调用 power(2, 3)的求值过程



√ 自测题

12. 以下程序输出什么？

```
#include <iostream>
using namespace std;
int mystery(int n);
// 前条件：n >= 1

int main()
{
    cout << mystery(3);
    return 0;
}
int mystery(int n)
{
    if (n <= 1)
        return 1;
    else
        return ( mystery(n - 1) + n );
}
```

13. 以下程序输出什么？rose 是哪一个著名的数学函数？

```
#include <iostream>
using namespace std;
int rose(int n);
// 前条件：n >= 0

int main()
{
    cout << rose(4);
    return 0;
}

int rose(int n)
{
    if (n <= 0)
        return 1;
    else
        return ( rose(n - 1) * n );
}
```

14. 重新定义 power 函数，使其还能接受负指数。为此，必须将返回值的类型修改为 double。对于这个重新定义的 power 版本，它的函数声明和头部注释如下所示：

```
double power(int x, int n);
// 前条件：如果 n < 0，那么 x 不为 0
// 返回 x 的 n 次方
```

提示：$x^{-n} = 1/(x^n)$。

14.3　递归思想

世上只有两种人，一种把世界分成两种人，一种不会。

——*无名氏*

递归设计技术

定义和使用递归函数时，不需要一直监视栈和被暂停的计算。递归的强大优势在于完

全可以忽略这些细节，让计算机自动完成所有繁琐的跟踪记录工作。以图 14.3 的 power 函数为例，可像下面这样想象它的定义：

power(x, n)返回 power(x, n - 1) * x

由于 x^n 等于 $x^{n-1} * x$，所以返回值是正确的，但前提是计算最终肯定会抵达停止情况，并正确计算停止情况。所以，在核实了递归部分的正确性之后，唯一还需核实的就是递归调用链，确保它肯定能抵达停止情况，而且停止情况肯定能返回一个正确的值。

设计递归函数时，不需要为那个函数的所有实例都完整地跟踪一遍递归调用链。如函数要返回值，唯一要做的就是核实它是否满足以下三个要求。

(1) 没有无穷递归(一个递归调用引发另一个递归调用，后者再引发另一个，依此类推。但这种调用链最终都必须遇到一种停止情况，否则会产生无穷递归)。
(2) 每种停止情况都能返回那种情况下的正确值。
(3) 对于涉及递归的每一种情况，如所有递归调用都返回正确的值，函数返回的终值就是正确的。

例如图 14.3 中的 power 函数，它就满足所有这些要求。

(1) **没有无穷递归**：power(x, n)的第二个参数在每次递归调用时都递减 1，递归调用链最终会抵达 power(x, 0)这个停止情况。所以，没有无穷递归。
(2) **每种停止情况都返回那种情况下的正确值**：唯一停止情况就是 power(x, 0)。power(x, 0)这种形式的调用肯定返回 1，x^0 的正确值是 1。所以，停止情况返回正确值。
(3) **对于涉及递归的每一种情况，如所有递归调用都返回正确的值，函数返回的终值就是正确的**：涉及递归的唯一情况就是 n > 1。在 n > 1 的情况下，power(x, n)返回：

    power(x, n - 1) * x

    为了证明这是正确的返回值，注意假定 power(x, n-1)返回正确的值，那么 power(x, n-1)返回 x^{n-1}，所以power(x, n)返回以下结果：

    $x^{n-1} * x$，也就是 x^n

    这正是 power(x, n)的正确值。

为确保 power 的定义正确，上面就是唯一要检查的内容(这种技术称为**数学归纳法**，你上数学课时可能听说过。但为了使用这种技术，并不要求真的熟悉该术语)。

前面为你提供了三个条件来检验返回值的递归函数是否正确。大致相同的规则也适用于递归的 void 函数。如果能证明递归的 void 函数定义满足以下三个条件，void 函数肯定能正确工作。

(1) 没有无穷递归。
(2) 每种停止情况都执行那种情况下正确的操作。
(3) 针对涉及递归的每一种情况，如所有递归调用都正确执行它们的操作，那么整个情况都会正确执行。

案例分析：二叉搜索(递归思想示例)

本案例要开发一个递归函数，它搜索数组，判断其中是否包含指定值。例如，数组可能包含一个无效信用卡号列表。商店店员要搜索该列表，判断顾客信用卡是否有效。第 7 章的图 7.10 讨论了搜索数组的简单方法，它单纯检查每个数组元素。本节要开发另一种方法来搜索已排序数组，速度快得多。

数组 a 的索引范围在 0～finalIndex 之间。为简化搜索数组的任务，假定数组已排好序，也就是假定：

```
a[0] <= a[1] <= a[2] <= … <= a[finalIndex]
```

搜索数组时，可能既想知道值是否在列表中，又想知道具体在哪个位置。例如，假定要搜索信用卡号，那么数组索引可作为记录编号使用。在另一个数组的相同索引处，可能包含一个电话号码或其他信息，操作人员可利用这些信息向信用卡管理部门报告可疑信用卡。所以，假如在数组中搜索到了目标值，就希望函数指出那个值在数组中的具体位置。

1. 问题定义

准备让函数通过两个传引用参数返回搜索结果。一个参数是 bool 类型的 found。如发现值，则 found 设为 true。发现值后，另一个参数(名为 location)设为发现的那个值的索引。用 key 表示要搜索的值，要完成的任务可精确定义如下。

前条件：a[0]～a[finalIndex]按升序排列。

后条件：如果 key 不是 a[0]～a[finalIndex]的某个值，那么 found == false；否则，a[location] == key，而 found == true。

2. 算法设计

现在要设计解决该问题的算法。可设想一个非常具体的情况以加深体会。假定无效卡号非常多，以至于要用一本书列出它们。事实上，对那些没有配备电脑的商店，无效信用卡的卡号正是以这种方式分发给它们的。假定你是店员，并从顾客手中接过了一张信用卡。现在，必须检查它的号码是否在列表中(在列表中表明该卡无效)。那么，具体应该怎样操作呢？首先翻到书的中央，检查号码是否在那里。如果不在，而且小于中间的号码，就向前翻，一直翻到这本书的开头。如果号码大于中间的号码，则向后翻，一直翻到这本书的末尾。下面是基于该思路的第一个算法草稿：

```
found = false; // 就目前来说
mid = 0～finalIndex 的一个大致的中点;
if (key == a[mid])
{
    found = true;
    location = mid;
}
else if (key < a[mid])
    搜索 a[0]～a[mid - 1];
else if (key > a[mid])
    搜索 a[mid + 1]～a[finalIndex];
```

由于搜索较短的列表是当前这个任务的一个较小的版本，所以自然应该使用递归算法。

通过递归调用同一个算法，需搜索的列表会变得越来越小。

当然，上述伪代码显得过于粗略，不太好转换成 C++代码。问题和递归调用有关。其中包含以下两个递归调用：

```
搜索 a[0]～a[mid - 1];
搜索 a[mid + 1]～a[finalIndex];
```

实现递归调用还需要两个参数。递归调用指定要搜索的数组的一个子范围。在一种情况下，该子范围是索引 0～索引 mid - 1 的元素。在另一种情况下，是索引 mid + 1～finalIndex 的元素。两个额外的参数将指定搜索的第一个索引和最后一个索引，所以将这两个参数命名为 first 和 last。用这两个参数来表示最小索引和最大索引，而不是使用 0 和 finalIndex，伪代码就可以更准确地表示成如下形式：

```
为了搜索 a[first]～a[last]，要做以下事情：
found = false; // 就目前来说
mid = first～last 的一个大致的中点;
if (key == a[mid])
{
    found = true;
    location = mid;
}
else if (key < a[mid])
    搜索 a[first]～a[mid - 1];
else if (key > a[mid])
    搜索 a[mid + 1]～a[last];
```

为了搜索整个数组，要将 first 设置成 0，将 last 设置成 finalIndex，再执行算法。执行递归调用时，则要将 first 和 last 设置成其他值。例如，第一个递归调用将 first 设置成 0，将 last 设置成 mid -1。

和任何递归算法一样，必须确保算法最后能终止，不会产生无穷递归。在列表中发现要查找的值，就不再产生新的递归调用，整个过程终止。但需要以某种方式检测数字不在列表中的情况。每次递归调用时，要么 first 的值递增，要么 last 的值递减。如果它们彼此相交，而且 first 变得比 last 还要大，表明没有更多索引需要检查，也表明 key 值不在数组中。将这个测试添加到伪代码中，就获得了如图 14.5 所示的完整方案。

图 14.5 二叉搜索的伪代码

```
int a[Some_Size_Value];
```

搜索 a[first]～a[last]的算法

```
1     // 前条件:
2     // a[first] <= a[first + 1] <= a[first + 2] <= … <= a[last]
```

为了定位 key 值：

```
1     if (first > last) // 一种停止情况
2         found = false;
3     else
4     {
5         mid = first～last 的一个大致的中点;
6         if (key == a[mid]) // 一种停止情况
7         {
8             found = true;
9             location = mid;
10        }
```

```
11        else if (key < a[mid]) // 一种递归情况
12           搜索 a[first]~a[mid - 1];
13        else if (key > a[mid]) // 一种递归情况
14           搜索 a[mid + 1]~a[last];
15     }
```

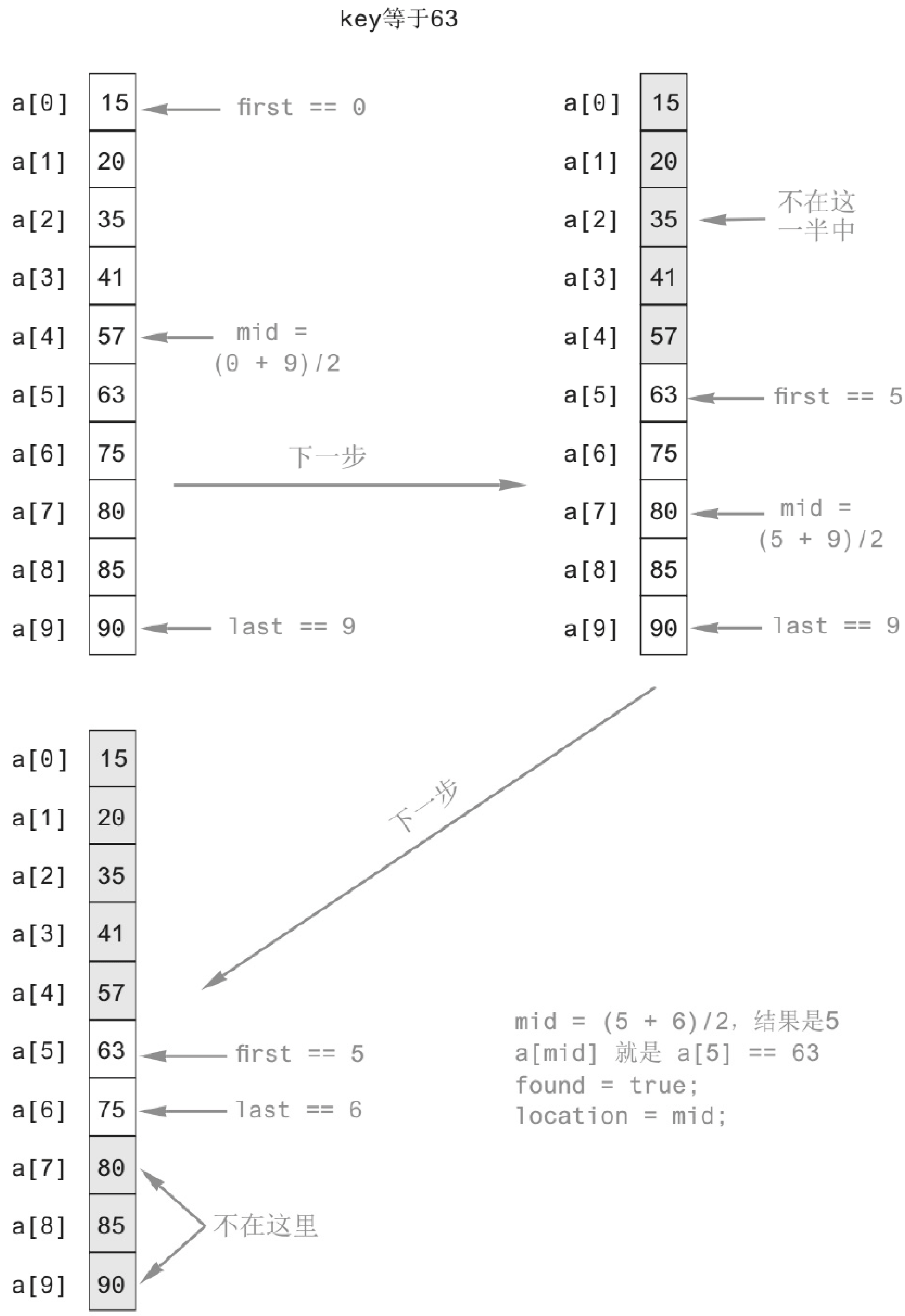
3. 编码

现在可将伪代码直接转换成 C++代码，图 14.6 展示了最终结果。search 函数是图 14.5 的递归算法的一个实现。图 14.7 图示了为示范数组执行该函数的过程。

图 14.6　二叉搜索递归函数

```
1  // 该程序演示用于二叉搜索的递归函数
2  #include <iostream>
3  using namespace std;
4  const int ARRAY_SIZE = 10;
5
6
7  void search(const int a[], int first, int last,
8              int key, bool& found, int& location);
9  // 前条件：a[first]~a[last]按升序排列
10 // 后条件：如果 key 不是 a[first]~a[last]的某个值，那么 found == false；
11 // 否则，a[location] == key，而 found == true
12
13
14 int main()
15 {
16     int a[ARRAY_SIZE];
17     const int finalIndex = ARRAY_SIZE - 1;
18
    <这个部分包含了一些用于填充和排序数组 a 的代码。它们的细节与本例无关>
19     int key, location;
20     bool found;
21     cout << "Enter number to be located: ";
22     cin >> key;
23     search(a, 0, finalIndex, key, found, location);
24
25     if (found)
26         cout << key << " is in index location "
27              << location << endl;
28     else
29         cout << key << " is not in the array." << endl;
30
31     return 0;
32 }
33 void search(const int a[], int first, int last,
34                           int key, bool& found, int& location)
35 {
36     int mid;
37     if (first > last)
38     {
39         found = false;
40     }
41     else
42     {
43         mid = (first + last)/2;
44
45         if (key == a[mid])
46         {
47             found = true;
48             location = mid;
49         }
50         else if (key < a[mid])
51         {
52             search(a, first, mid - 1, key, found, location);
53         }
54         else if (key > a[mid])
55         {
56             search(a, mid + 1, last, key, found, location);
57         }
58     }
59 }
```

图 14.7　search 函数的搜索过程



注意，search 函数解决了一个比原始任务更常规的问题。我们的目标是设计函数来搜索整个数组。但函数允许搜索数组任何一个区段，只需指定索引边界 first 和 last。设计递归函数时，这是常见的设计手法。很多情况下都需要解决更常规的问题，才能正确表示递归算法。本例只希望获得在 first 等于 0，而 last 等于 finalIndex 的情况下的答案。但递归调用会将 first 和 last 设置成除 0 和 finalIndex 之外的值。

4．检查递归

上一节给出了验证递归 void 函数定义是否正确的三个条件。下面针对图 14.6 的 search 函数核实这些条件。

(1) **没有无穷递归**。在每个递归调用中，要么 first 的值递增，要么 last 的值递减。如递归调用链不以其他方式终止，那么函数调用最终会出现 first 比 last 大的情况，这是一种停止情况。

(2) **每种停止情况都执行那种情况下的正确操作**。有两种停止情况，要么是 first > last，要么是 key == a[mid]。下面分别解释每种情况。

- 如 first > last，表明 a[first]和 a[last]之间没有更多的数组元素，所以 key 不在数组的这个区段中(事实上，这个数组区段什么都没有)。所以，如 first > last，search 函数会将 found 正确设置成 false。
- 如 key == a[mid]，算法会正确设置 found 和 location，使 found 等于 true，并使 location 等于 mid。因此，两种停止情况都正确。

(3) **对于涉及递归的每一种情况，如所有递归调用都正确执行它们的操作，那么整个情况都会得以正确执行**。有两种情况会产生递归调用。一种是 key < a[mid]，另一种是 key > a[mid]。需检查这两种情况。

首先假定 key < a[mid]。这种情况下，由于数组已排序，所以我们知道如果 key 在数组中的某个地方，key 必然是 a[first]～a[mid - 1]的某个元素。因此，函数只需搜索这些元素，这恰好是以下递归调用所做的事情：

```
search(a, first, mid - 1, key, found, location);
```

所以，如递归调用正确，整个操作都正确。

接着假定 key > a[mid]。这种情况下，由于数组已排序，所以我们知道如果 key 在数组中某个地方，那么 key 必然是 a[mid + 1]～a[last]的某个元素。因此，函数只需搜索这些元素，这恰好是以下递归调用所做的事情：

```
search(a, mid + 1, last, key, found, location);
```

所以，如递归调用正确，整个操作都正确。

总之，在上述两种情况下，函数都会执行正确的操作(假定递归调用执行正确的操作)。

search 函数通过了全部三项测试，表明它是一个良好的、正确的递归函数定义。

5. 效率

和顺序搜索算法相比，二叉搜索算法快得多。在二叉搜索中，一开始就将搜索范围缩小至原数组一半左右。然后继续拆分，将范围缩小至数组的四分之一，再是八分之一，依此类推。这样就获得了一个速度极快的算法。对于 100 个元素的数组，二叉搜索最多只需将 key 值与 7 个数组元素比较。相反，顺序搜索最多可能要拿全部 100 个元素与 key 值比较，平均也需比较 50 个元素。此外，数组越大，二叉搜索效率越显著。对于 1000 个元素的数组，二叉搜索只需将 key 值与大约 10 个数组元素比较；相反，顺序搜索算法平均要比较 500 个元素。

图 14.8 展示了 search 函数的迭代版本。某些系统上，迭代版本的资源利用率比递归版本更好。迭代版本的算法直接从递归版本推导。在迭代版本中，局部变量 first 和 last 镜像了递归版本的同名参数的角色。如本例所示，即使以后要转换成迭代版本，最好也是先归纳好递归算法。

图 14.8　二叉搜索的迭代版本

函数声明

```
1  void search(const int a[], int lowEnd, int highEnd,
2                         int key, bool& found, int& location);
3  // 前条件：a[lowEnd]～a[highEnd]按升序排列
4  // 后条件：如果 key 不是 a[lowEnd]～a[highEnd]的某个值，那么 found == false；
5  // 否则，a[location] == key，而 found == true
```

函数定义

```
1  void search( const int a[], int lowEnd, int highEnd,
2                  int key, bool& found, int& location)
3  {
4      int first = lowEnd;
5      int last = highEnd;
6      int mid;
7
8      found = false; // 就目前来说
9      while ( (first <= last) && !(found) )
10     {
11         mid = (first + last)/2;
12         if (key == a[mid])
13         {
14             found = true;
15             location = mid;
16         }
17         else if (key < a[mid])
18         {
19             last = mid - 1;
20         }
21         else if (key > a[mid])
22         {
23             first = mid + 1;
24         }
25     }
26 }
```

编程实例　递归成员函数

类的成员函数可以是递归的。成员函数可采取与普通函数相同的方式使用递归。图 14.9 包含了递归成员函数的例子，其中使用的 BankAccount 类与图 10.6 定义的 BankAccount 类相同，只是重载了成员函数名称 update。update 的第一个版本无参，直接在银行账户余额上加 1 年的单利。update 的另一个(新)版本则要获取代表年数的 int 参数，它更新账户，在余额上添加那么多年的利息。新版本 update 是递归的。新的 update 函数有一个名为 years 的参数。它的算法如下：

```
如果 years 等于 1，那么 // 停止情况：
    调用另一个 update 函数(无参的那个)。
如果 years 大于 1，那么 // 递归情况：
    执行一个递归调用，产生 years - 1 年的利息，然后调用
    另一个 update 函数(无参的那个)，再增加 1 年的利息
```

核实之前罗列的三个条件，很容易就能证明上述算法能产生正确结果。

(1) **没有无穷递归**。每个递归调用都使 years 递减 1，直至 years 最终变成 1，这是停止情况。所以，不会出现无穷递归的情况。

(2) **每种停止情况都执行那种情况下的正确操作**。一种停止情况是 years == 1。该情况能产生正确行动，因为它只是调用另一个名为 update 的重载成员函数。第 10 章已验证了那个函数的正确性。

(3) **对于涉及递归的每一种情况。如所有递归调用都正确执行它们的操作，那么整个情况都会正确执行：**递归情况(years > 1)能正确工作，因为一旦递归调用正确加上 years -1 年的利息，剩下的唯一操作就是再加 1 年的利息。调用 update 的无参数重载版本，即可正确加 1 年的利息。所以，如果递归调用执行正确的行动，那么 years > 1 情况下的整个行动都是正确的。

图 14.9　一个递归成员函数

这个程序的 BankAccount 类是图 10.6 的 BankAccount 类的改进版本

同名的两个不同的函数（指第 16 行和第 18 行）

```
1  // 该程序演示递归成员函数 update(years)
2  #include <iostream>
3  using namespace std;
4
5  // 代表一个银行账户的类:
6  class BankAccount
7  {
8  public:
9      BankAccount(int dollars, int cents, double rate);
10     //将余额初始化为$dollars.cents，将利率初始化为利率百分比
11     BankAccount(int dollars, double rate);
12     // 将账户余额初始化为$dollars.00，将利率初始化为利率百分比
13
14     BankAccount();
15     // 将账户余额初始化为$0.00，将利率初始化为 0.0%
16     void update();
17     // 后条件：将 1 年的单利添加到账户余额上
18     void update(int years);
19     // 后条件：将指定年数的利息加到账户余额上。利息每年都要复合
20     double getBalance();
21     // 返回当前账户余额
22     double getRate();
23     // 将当前账户利率作为一个百分比返回
24     void output(ostream& outs);
25     // 前条件：如果 outs 是一个文件输出流，那么 outs 已经连接到一个文件
26     // 后条件：账户余额和利率写入 outs 流
27 private:
28     double balance;
29     double interestRate;
30     double fraction(double percent); // 将百分比转换成小数
31 };
32
33 int main()
34 {
35     BankAccount yourAccount(100, 5);
36     yourAccount.update(10);
37     cout.setf(ios::fixed);
38     cout.setf(ios::showpoint);
39     cout.precision(2);
40     cout << "If you deposit $100.00 at 5% interest, then\n"
41          << "in ten years your account will be worth $"
42          << yourAccount.getBalance() << endl;
43     return 0;
44 }
45
46 void BankAccount::update()
47 {
48     balance = balance + fraction(interestRate)*balance;
48 }
49
50 void BankAccount::update(int years)
51 {
52     if (years == 1)
53     {
```

```
54          update();
55      }
56      else if (years > 1)
57      {
58          update(years - 1);
59          update();
60      }
61  }
```

重载(也就是调用同名的另一个函数)

递归函数调用

<其他成员函数定义与图 10.5 和图 10.6 相同。如果只是为了理解这个例子，不需要阅读那些定义>

示范对话

```
If you deposit $100.00 at 5% interest, then
in ten years your account will be worth $162.89
```

本例重载了 update。所以，现在有两个不同的 update 函数：一个无参，一个要获取单个参数。不要混淆对这两个同名函数的调用。它们是两个不同的函数，而且从编译器的角度看，两者只是凑巧同名。在接收单个参数的 update 函数的定义中，虽然包含了对无参 update 的调用，但不能算递归调用。只有调用函数声明完全相同的 update，才算是递归调用。为理解这个问题，可将无参 update 重命名为无参 postOneYear()，而不是仍然称之为 update()。此时递归版本的 update 的定义变成：

```
void BankAccount::update(int years)
{
    if (years == 1)
    {
        postOneYear();
    }
    else if (years > 1)
    {
        update(years - 1);
        postOneYear();
    }
}
```

递归和重载

不要混淆递归(recursion)和重载(overloading)。重载函数名称会获得同名的两个不同函数。其中一个函数的定义包括对另一个函数的调用，不算是递归。在递归函数定义中，应包括对完全相同的一个函数的调用，那个函数必须具有完全相同的定义，而不能仅仅是同名。混淆递归和重载不算严重错误，因为两者都合法。但应该清楚地理解自己所说的术语，以利于和其他程序员沟通，这要求你掌握术语背后的基本原理。

√ 自测题

15. 写以下递归函数定义：

```
int squares(int n);
// 前条件: n >= 1
// 返回 1~n 的各个数字的平方和
```

例如，squares(3)应该返回 14，因 $1^2 + 2^2 + 3^2$ 等于 14。

16. 为图 14.9 的单参数成员函数 BankAccount::update(*int* years)写迭代版本。

小　结

- 如果一个问题能缩小成同一个问题的较小版本，那么使用递归方案也许能更轻松地发现和实现解决方案。
- 函数定义的递归算法通常包含两种情况：在一种或更多情况下，包含了至少一个递归调用；另外还有一种或更多停止情况。在停止情况下，问题会得到解决，不需要任何递归调用。
- 写递归函数定义时，一定要核实函数不会产生无穷递归。
- 定义递归函数时，要参照 14.3 节罗列的三个条件核实函数的正确性。
- 设计递归函数完成任务，通常有必要解决一个比给定任务更常规的问题。为了产生正确的递归调用，这也许是必须的，因为较小的问题可能和给定任务不完全一致。例如在二叉搜索问题中，虽然任务是搜索整个数组，但递归方案对其进行了常规化，能搜索数组的任何一部分(既可搜索整个数组，也可搜索其中一部分)。

自测题答案

1. Hip Hip Hurray

2.
```
void stars(int n)
{
    cout << '*';
    if (n > 1)
        stars(n - 1);
}
```

下面这个版本也正确，只是稍微复杂一些：

```
void stars(int n)
{
    if (n <= 1)
    {
        cout << '*';
    }
    else
    {
        stars(n - 1);
        cout << '*';
    }
}
```

3.
```
void backward(int n)
{
    if (n < 10)
    {
        cout << n;
    }
    else
    {
        cout << (n % 10); // 输出最后一个数位
        backward(n / 10); // 反方向输出其他数位
    }
}
```

4 和 5. 自测题 4 的答案是 writeUp(**int** n)，自测题 5 的答案是 writeDown(**int** n)。

```
#include <iostream>
using namespace std;
void writeDown( int n)
{
    if (n >= 1)
    {
        cout << n << " ";
        writeDown(n - 1);
    }
}

void writeUp(int n)
{
    if (n >= 1)
    {
        writeUp(n - 1);
        cout << n << " ";
    }
}

// 对自测题 4 和自测题 5 进行测试的代码
int main()
{
   cout << "calling writeUp(" << 10 << ")\n";
   writeUp(10);
```

```
    cout << endl;
    cout << "calling writeDown(" << 10 << ")\n";
    writeDown(10);
    cout << endl;
    return 0;
}
/* 测试结果
calling writeUp(10)
1 2 3 4 5 6 7 8 9 10
calling writeDown(10)
10 9 8 7 6 5 4 3 2 1
*/
```

6. 栈溢出表明计算机试图在栈中放入太多活动帧，超出了系统限制。一个可能的原因是无穷递归。

7.
```
void cheers(int n)
{
    while (n > 1)
    {
        cout << "Hip ";
        n--;
    }
    cout << "Hurray\n";
}
```

8.
```
void stars(int n)
{
    for (int count = 1; count <= n; count++)
        cout << '*';
}
```

9.
```
void backward(int n)
{
    while (n >= 10)
    {
        cout << (n%10); // 输出最后一个数位
        n = n/10; // 丢弃最后一个数位
    }
    cout << n;
}
```

10. 跟踪自测题 4：

如果 n = 3，则执行以下代码：

```
if (3 >= 1)
{
        writeUp(3 - 1);
        cout << 3 << " ";
}
```

在接下来的递归中，n = 2；执行的代码如下：

```
if (2 >= 1)
{
        writeUp(2 - 1);
        cout << 2 << " ";
}
```

在接下来的递归中，n = 1；执行的代码如下：

```
if (1 >= 1)
{
        writeUp(1 - 1);
        cout << 1 << " ";
}
```

在最后一个递归中，n = 0；执行的代码如下：

```
if (0 >= 1) // 条件为 false，跳过主体
```

```
{
        // 跳过
}
```

递归解链；输出是 1 2 3(注意，递归期间就已输出，不是在解链过程中输出)。

11. 跟踪自测题 5：

如果 n = 3，要执行的代码如下：

```
if (3 >= 1)
{
        cout << 3 << " ";
        writeDown(3 - 1);
}
```

在接下来的递归中，n = 2；要执行的代码如下：

```
if (2 >= 1)
{
        cout << 2 << " ";
        writeDown(2 - 1);
}
```

在接下来的递归中，n = 1；要执行的代码如下：

```
if (1 >= 1)
{
        cout << 1 << " ";
        writeDown(1 - 1);
}
```

在最后一个递归中，n = 0，条件表达式为 false，所以跳过 if 语句的主体：

```
if (0 >= 1)  // 条件为 false
{
        // 跳过
}
```

输出 3 2 1。

12. 输出 6。

13. 输出 24。这是阶乘函数，在数学中表示成 $n!$，定义如下：

$$n! = n * (n - 1) * (n - 2) * \cdots * 1$$

14.
```
// 使用 iostream 和 cstdlib:
double power(int x, int n)
{
    if (n < 0 && x == 0)
    {
        cout << "Illegal argument to power.\n";
        exit(1);
    }

    if (n < 0)
        return ( 1/power(x, -n));
    else if (n > 0)
        return ( power(x, n - 1)*x );
    else // n == 0
        return(1.0);
}
```

15.
```
int squares(int n)
{
    if (n <= 1)
        return 1;
    else
        return ( squares(n - 1) + n*n );
}
```

16.
```
void BankAccount::update(int years)
{
    for (int count = 1; count <= years; count++)
        update();
}
```

编 程 练 习

编程练习一般只需写很小的程序，运用本章提到的编程概念。

1. 写递归函数定义，该函数获取一个 int 类型的参数 n，返回第 n 个斐波那契数字。参见第 3 章的编程项目 6 了解斐波那契数字的定义。将函数嵌入程序来测试。
2. 为第 7 章图 7.12 的排序程序中使用的 indexOfSmallest 函数写递归版本。将函数嵌入程序来测试。
3. 为第 7 章图 7.10 的 search 函数写递归版本。
4. 房间有 n 个人，n 是大于等于 2 的整数。每个人都和其他每个人握一次手。房间里发生的握手总次数是多少？写递归函数解决问题，函数头如下：

   ```
   int handshake(int n)
   ```

   handshake(n) 返回房间里 n 个人的握手总次数。**提示：**如房间只有一个或两个人，那么：

   ```
   handshake(1)  = 0
   handshake(2)  = 1
   ```

5. 写递归函数，如输入字符串是回文就返回 true，否则返回 false。可检查第一个字符是否等于最后一个字符。如果是，就进行递归调用，在输入字符串中剔除头尾字符。必须定义合适的停止情况。用几个回文和非回文测试函数。

编 程 项 目

编程项目要求综合运用多方面的知识来解决问题，程序一般比编程练习大，解题方式多样化。

1. 假定总共 n 样东西，从中选择 r 样不同的东西。为计算总共有多少种选择方式，可使用以下公式：

   $$C(n,r) = n!/(r!*(n-r)!)$$

   阶乘函数 $n!$ 的定义如下：

   $$n! = n*(n-1)*(n-2)*\cdots\cdots*1$$

   归纳上述公式的递归版本，写递归函数来计算公式的值。将函数嵌入程序并测试。

2. 写递归函数，一个参数是字符数组，另两个是数组索引边界。函数反转两个边界之间的元素的顺序。例如，对于以下数组：

   ```
   a[0] == 'A' a[1] == 'B' a[2] == 'C' a[3] == 'D' a[4] == 'E'
   ```

   而且边界是 1 和 4，函数运行后，数组元素应变成以下形式：

   ```
   a[0] == 'A' a[1] == 'E' a[2] == 'D' a[3] == 'C' a[4] == 'B
   ```

   将函数嵌入程序并测试。完全调试好函数后，再定义一个函数，它只有一个参数，也就是包含了一个字符串值的数组。函数反转数组参数中字符串值的拼写。该函数要包含对之前的递归函数的调用。将第二个函数嵌入程序并测试。

3. 为编程项目 1 的递归函数写迭代版本。嵌入程序并测试。
4. 视频讲解：*Solution to Programming Project 14.4*

写递归函数，根据以下思路按升序排序一个整数数组：最小元素放到第一个位置，再通过一个递归调用对数组剩余部分排序。这是第 7 章讨论的选择排序算法的递归版本(**注意**：直接拿第 7 章的程序并在其中插入 `indexOfSmallest` 的递归版本是不够的。执行排序的函数本身必须是递归的，不能只是使用一个递归函数)。

5. 汉诺塔问题：传说有一些僧人，他们为一个涉及 64 个石盘的难题困惑不已。僧人的目的是将全部石盘从一个柱子移动到另一个柱子，中间能够借助第三个柱子。一旦完成，时间就会停止(世界末日就会来临)。现在，将世界末日(因为时间会停止)和宗教问题留给那些更有资格的人士。我们感兴趣的是如何用递归方案解决这个问题。

   总共 n 个石盘叠放在其中一个柱子上，它们的直径从上到下逐渐增加。你的任务是每次将一个盘子从一个柱子移动到另一个柱子。注意，任何盘子都能从任何柱子移动到其他任何一个柱子，唯一规则是不能将较大的盘子放在较小的盘子上。第三个柱子的存在，使解决这个问题变得可能。你的任务是写递归函数来描述解决方案。没有学习图形显示，所以应该输出一组文字指令解决这个问题。

   提示：如果先将 n-1 个盘子从第一个柱子移动到第三个柱子，使用第二个柱子作为临时存放点，最后一个盘子就能从第一个柱子移动到第二个柱子。然后，可采用相同的技术，将 n-1 个盘子从第三个柱子移动到二个柱子，使用第一个柱子作为临时存放点。看，困扰僧人许久的问题就这样被你解决了！你只需要确定什么是非递归情况，什么是递归情况，以及在什么时候输出移动盘子的指令。

6. Jump It 游戏棋盘上，一行有 n 个正整数，但第一列始终为 0。这些数字代表要进入这一列所需要的花销，下面是一个 n 等于 6 时的游戏棋盘：

0	3	80	6	57	10

   游戏目标是以最低的总花销从第一列移到最后一列。每一列的数字代表进入那一列所需的花销。游戏始终从第 1 列开始，并且有两种走法。要么移到相邻的那一列，要么跳过相邻的那一列，移到它的下一列。玩一次的总花销就是去过的所有列的花销之和。

   在如上所示的棋盘中，有好几种方法可以走到最后一列。从第 1 列开始，当前花销为 0。然后，可以跳到 80，再跳到 57，再跳到 10，得到总花销为 80＋57＋10＝147。然而，一个更便宜的路线是跳到 3，再跳到 6，最后跳到 10，得到总花销为 3＋6＋10＝19。

   针对这个问题写一个递归解决方案，为一个任意大的棋盘(用一个数组表示)算出游戏的最低花销，并输出此值。你的程序不必输出实际的跳转顺序，只需输出这个棋盘的最低花销。用更大的棋盘和更大的 n 值测试你编写的程序，从中体会程序的灵活性和效率。

7. 假定我们可以从自动售货机购买巧克力，每块 1 美元。每买一块都返一张优惠券。可以再用 7 张优惠券从自动售货机中兑换一块巧克力。我们想知道的是：花 n 美元一共能吃到多少块巧克力，包括用优惠券兑换的。

   例如，如果有 20 美元，最开始用这 20 美元购买 20 块巧克力。这样就可获得 20 张优惠券。14 张优惠券可兑换 2 块巧克力。这两块巧克力又含有两张优惠券，所以，再加上最初剩下的 6 张优惠券，就一共有 8 张优惠券。这样一来，又可以用其中的 7 张优惠券兑换最后一块巧克力。最后，我们得到 23 块巧克力和两张富余的优惠券。

   针对这个问题写一个递归解决方案，要求用户指定准备花多少美元购买巧克力，然后输出花完这些钱后所得到的巧克力块数，并尽量多地兑换完优惠券。你的递归函数将基于拥有的优惠券的数量。

8. 有的问题要求找出一组项目的所有排列与组合。对于有 n 个项的一个集合$\{ a_1, a_2, a_3, \ldots a_n \}$，排列组合的总数是 $n!$种。例如，如果给定集合$\{ 1, 2, 3 \}$，那么共有 6 种排列组合：

   $$\{3, 2, 1\}\ \{2,3,1\}\ \{2,1,3\}\{3,1,2\}\{1,3,2\}\ \{1,2,3\}$$

   写递归函数来生成一个数字集合的所有排列组合。下面会给出一个解决方案的思路，但具体怎么实现要取决于你。数字的排列组合可以用多种方式来存储，比如节点链表、向量链表和数组等，请自行选择一种。你的程序应针对多个不同大小的集合调用递归函数，打印每个集合的所有排列组合。

   一个解决方案是先拿掉集合中的第 n 个项。针对由$(n - 1)$个项构成的集合，递归地找出所有排列组合。

在每个排列组合的每一个位置，都插入刚才被拿掉的第 n 个项，这样就得到了包含第 n 个项的一组新的排列组合。“终止情况”在集合中只有一个项的时候发生，此时只需处理一个项的排列组合。

例如，假定要计算{1,2,3}的所有排列组合。先把 3 拿掉，递归地找出{1,2}的所有排列组合。这些排列组合如下：

{1,2}　　　　　{2,1}

接着在每个排列组合的每个位置都插入 3。对于第一个排列组合，3 的插入位置是 1 之前，1 和 2 之间，以及 2 之后；对于第二个排列组合，3 的插入位置是 2 之前，2 和 1 之间，以及 1 之后：

{3,1,2}　{1,3,2}　{1,2,3}　{3,2,1}　{2,3,1}　{2,1,3}

最后的 6 个排列组合即是{1,2,3}的所有排列组合。

9. “词梯”(word ladder)是刘易斯・卡罗尔于 1877 年发明的一种单词游戏。该游戏有一个起始词和一个终止词，游戏者需要发现一条连接两个词的词汇链，词汇链上的两个相邻词只差一个字母。玩家一般是更改起始词中的一个字母，获得一个新词，再继续更改新词中的某个字母，再获得一个新词，最终获得终止词。要求每个词都必须是英语单词。

   例如，从 FISH 开始可建立一个到达 MAST 的词梯：

   FISH, WISH, WASH, MASH, MAST

   写程序递归查找给定起始词和终止词的词梯，或报告不存在词梯。使用本书配套资源中的 words.txt 文件来作为有效词字典。文件包含 87314 个单词。不要求程序找出最短词梯，存在的任何词梯都可以。

第 15 章　继承

最温暖舒适的环境之中，……

——威廉·莎士比亚，《亨利四世》第 3 幕第 1 场

概述

面向对象编程(Object-Oriented Programming，OOP)是一种流行、强大的编程技术。它提供了“继承”这种抽象方式。意味着可以先定义并编译常规形式的类，再定义那个类的更具体版本，并继承前一个类的所有功能。C++的各种版本都提供了对继承的支持。

预备知识

15.1 节基于第 2 章～第 8 章、第 10 章～第 12 章的知识。15.2 节和 15.3 节不仅要基于第 2 章～第 8 章、第 10 章～第 12 章以及 15.1 节的知识，还要基于第 9 章和第 13 章的知识。

15.1　继 承 基 础

如果希望改变孩子的品格，我们首先应该检讨自己。看看自己有什么需要改变的。[①]

——卡尔·古斯塔夫·荣格[②](1875—1961)，《人格的整合》

C++最强大的功能之一就是通过继承从一个类派生出另一个类。**继承**是基于一个类(称为**基类**)创建新类(称为**派生类**)的过程。派生类自动拥有基类的所有成员变量和函数，并可根据需要添加更多的成员函数和/或成员变量。

第 10 章讲过，如果类 D 从类 B 派生，类 D 具有类 B 的所有功能，还可定义额外的。类 D 从类 B 派生，类 B 是基类，类 D 是派生类。还可以说 D 是**子类**，B 是**父类**。[③]

为了演示继承的用处，假定要设计一个智能家居系统，车库门和恒温器都已联网，可从计算机访问。系统要能控制和查看这些设备的状态(例如，门是否打开，恒温器是否设为 80 华氏度)。为不同设备使用统一接口能简化开发。继承使我们能做到这一点，还能良好地组织代码，避免代码重复。

首先想好智能家居系统中的各种设备的常规概念。每种设备都肯定有型号和序列号。或许每种设备都有一种查询状态的方式。我们把它建模为 `Device` 类，其中包含型号和序列号变量，还有一个函数供查询状态。基本思路是该类包含所有设备都适用的函数和属性。

然后考虑车库门。这是智能家居系统中的一种特定类型的设备。除了有型号、序列号和查询状态的方式，车库门设备还有一个特定的函数用于开关门。将车库门建模为 `DoorDevice` 类，并在其中添加 `openClose()` 函数。`DoorDevice` 类知道如何返回设备的状态。在常规 `Device` 类这一级，则无法获得返回特定设备的状态所需的信息，因为在那一级甚至不知道要处理的是哪种设备。虽然需要为 `DoorDevice` 添加函数来查询状态和

① 这里巧妙地用孩子指代继承类，用我们自己指代父类。希望孩子获得好的品格吗？先从自己做起吧！——译注

② 卡尔•古斯塔夫•荣格(Carl Gustav Jung)，瑞士心理学家、精神科医师，分析心理学的创始者。——译注

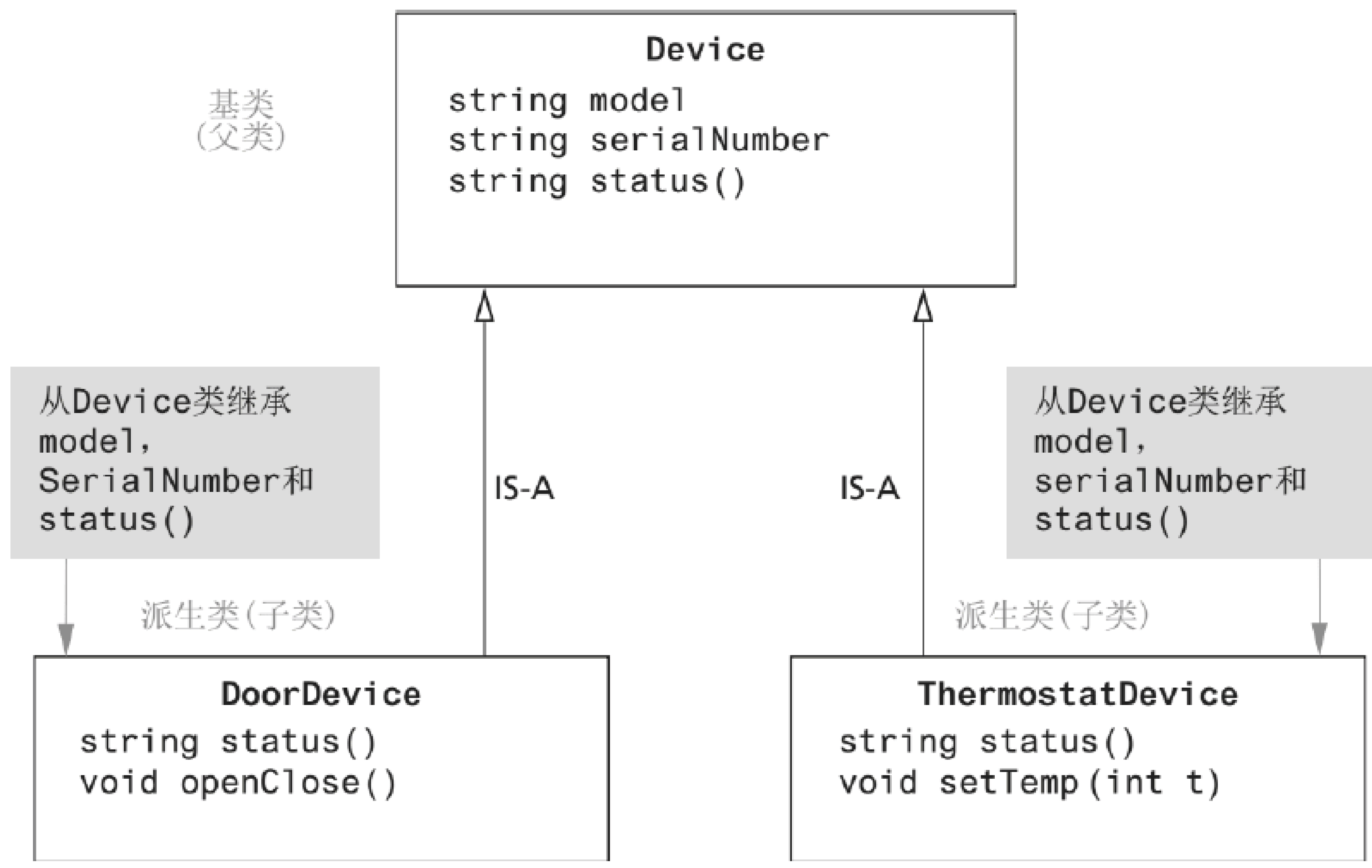
③ 一些作者将 D 称为子类，将 B 称为超类，而不是说派生类 D 和基类 B。然而，我们发现“派生类”和“基类”更不容易混淆。在阅读其他书时，请注意这些术语。

开/关门，但如果不重复为 Device 类写的用于处理型号/序列号的变量和代码就好了。

类似地，恒温器也有型号、序列号和查询状态的方式，并新增了用于设置温度的函数。可定义一个 ThermostatDevice 类，它提供了用于设置温度和返回设备状态的函数。同样地，如果不重复为 Device 类写的用于处理型号/序列号的变量和代码就好了。

解决方案就是继承。在本例中，DoorDevice“属于”(IS-A)Device，ThermostatDevice 也是。将 DoorDevice 和 ThermostatDevice 定义成 Device 的派生类，它们就能使用 Device 中定义好的方式访问型号和序列号，不需要重复这些代码。与此同时，可添加派生类特有的代码。图 15.1 展示了这些类之间的关系。

图 15.1　示范智能家居系统层次结构



定义好继承关系后，创建 DoorDevice 或 ThermostatDevice 类型的对象，就能访问 Device 定义好的函数和变量。例如，假定 thermostat 是 ThermostatDevice 类型的变量，就可访问 thermostat.model(要求 model 在 Device 类中是公共字符串变量)，省得重新定义 Device 类的代码和变量。

可修改 status()函数的工作方式。本章稍后会讲到，在派生类和基类中定义同一个函数时有两个选择，一个是重定义(redefine)，一个是重写(override)。本例想要重写。如果将 ThermostatDevice 类型的对象 thermostat 当作 Device 类型的对象处理(例如，将 thermostat 传给获取 Device 参数的函数)，在它上面调用 status()会调用 ThermostatDevice 的版本，而不是 Device 的。该行为对本例很重要，因为 Device 类不知道应返回什么“状态”。15.3 节将更深入地探讨这个主题。

第 10 章提到的定存账户也是继承有用的一个例子。我们提到定存账户是储蓄账户的更具体版本。从 SavingsAccount 派生出 CDAccount 类，创建 CDAccount 对象时将自动继承 SavingsAccount 的所有公共函数和变量。C++在预定义类中也使用了继承。用流执行文件 I/O 时，预定义类 ifstream 是从预定义类 istream 派生的，它添加了 open 和 close 等成员函数。流 cin 从属于代表“所有输入流”的类(istream 类)，不从属于代表“输入文件流”的类(ifstream 类)。一个证据是它没有派生类 ifstream 的成员函数 open 和 close。

派生类

假定要设计程序来记录合同工和计时工的信息。有一个自然的层次结构可组织这些类，因为所有这些人都具有作为员工的一些基本属性。

按小时计酬的员工是员工的子集。另一个子集是领固定月薪或周薪的员工。虽然程序可能不需要与所有员工的集合对应的一个类型，但假如泛化员工的概念，可能更有助于解决这个问题。例如，所有员工都有姓名和社会安全号。对于合同工和计时工，用于设置/更改姓名和社会安全号的成员函数是相同的。

可定义名为 Employee 的类，用它表示所有员工(无论合同工还是计时工)，再基于这个类定义计时工和合同工类。图 15.2 和图 15.3 展示了 Employee 类的定义。

图 15.2　基类 Employee 的接口

```
1  // 头文件 employee.h，它是 Employee 类的接口
2  // 该类主要作为基类使用，以便为不同类别的员工派生出相应的类
3  #ifndef EMPLOYEE_H
4  #define EMPLOYEE_H

5  #include <string>
6  using namespace std;

7  namespace employeessavitch
8  {

9    class Employee
10   {
11    public:
12        Employee();
13        Employee(string theName, string theSSN); // theSSN 是社会安全号
14        string getName() const;
15        string getSSN() const;
16        double getNetPay() const;
17        void setName(string newName);
18        void setSSN(string newSSN);
19        void setNetPay(double newNetPay);
20        void printCheck() const;
21     private:
22        string name;
23        string ssn;
24        double netPay;
25     };

26 } // employeessavitch

27 #endif // EMPLOYEE_H
```

图 15.3　基类 Employee 的实现

```
1  // employee.cpp 文件，它是 Employee 类的实现
2  // Employee 类的接口在头文件 employee.h 中
3  #include <string>
4  #include <cstdlib>
5  #include <iostream>
6  #include "employee.h"
7  using namespace std;

8  namespace employeessavitch
```

```
 9  {
10     Employee::Employee() : name("No name yet"), ssn("No number yet"), netPay(0)
11     {
12        // 主体有意留空
13     }
14     Employee::Employee(string theName, string theNumber)
15       : name(theName), ssn(theNumber), netPay(0)
16     {
17        // 主体有意留空
18     }
19     string Employee::getName() const
20     {
21        return name;
22     }
23     string Employee::getSSN() const
24     {
25        return ssn;
26     }
27
28     double Employee::getNetPay() const
29     {
30        return netPay;
31     }
32    void Employee::setName(string newName)
33     {
34        name = newName;
35     }
36     void Employee::setSSN(string newSSN)
37     {
38        ssn = newSSN;
39     }
40     void Employee::setNetPay (double newNetPay)
41     {
42        netPay = newNetPay;
43     }
44     void Employee::printCheck() const
45     {
46        cout << "\nERROR: printCheck FUNCTION CALLED FOR AN \n"
47             << "UNDIFFERENTIATED EMPLOYEE. Aborting the program.\n"
48             << "Check with the author of the program about this bug.\n";
49        exit(1);
50     }
51  } // employeessavitch
```

基于该定义可获得未细分的 Employee 对象。但之所以定义 Employee 类，目的就是为细分了类别的员工定义派生类。具体地说，printCheck(打印支票)函数的定义在不同派生类中肯定不一样，要为不同类别的员工生成不同形式的支票。Employee 类的 printCheck 函数(图 15.3)的定义反映了这一点。为没有细分的 Employee 打印支票没有意义，因为不知道该员工的工资细节。所以，为基类 Employee 的对象调用 printCheck 函数将会终止程序，并打印错误消息。后面会讲到，由于派生类拥有足够多的信息，所以能重定义 printCheck 函数，使其生成有意义的员工支票。

Employee 的派生类自动拥有 Employee 类的所有成员变量(name，ssn 和 netPay)和成员函数(printCheck，getName，setName 以及图 15.2 列出的其他成员函数)。我们说派生类**继承**了那些成员变量和成员函数。

图 15.4 和图 15.5 给出了 Employee 类的两个派生类的接口文件，其中包含相应的类定

义。图 15.4 定义 HourlyEmployee(计时工)类，图 15.5 定义 SalariedEmployee(合同工)类。我们将 Employee 类和两个派生类放在同一个命名空间。C++不要求它们在同一个命名空间。但由于是相互关联的类，所以最好如此。首先讨论图 15.4 的 HourlyEmployee 类。

注意，派生类的定义像其他任何类定义那样开始，只是在类定义第一行添加冒号、保留字 public 以及基类名称，如下所示(摘自图 15.4)：

```
class HourlyEmployee : public Employee
{
```

使用关键字 public 后，派生类(比如 HourlyEmployee)自动获得基类(比如 Employee)的所有成员变量和成员函数。可在派生类中添加新的成员变量和成员函数。

HourlyEmployee 类定义没有提到成员变量 name，ssn 和 netPay，但其每个对象都自动包含从 Employee 类继承的这些成员变量。HourlyEmployee 声明了另外两个成员变量：wageRate 和 hours。所以，HourlyEmployee 类的每个对象都有 5 个成员变量，即 name，ssn，netPay，wageRate 和 hours。注意派生类(比如 HourlyEmployee)的定义只列出新增成员变量。不需要列出基类定义的成员变量，它们自动提供给派生类。

除了从 Employee 继承所有成员变量，HourlyEmployee 类还继承了所有成员函数。因此，HourlyEmployee 类从 Employee 类继承成员函数 getName，getSSN，getNetPay，setName，setSSN，setNetPay 以及 printCheck。

继承的成员

派生类自动获得基类的所有成员变量和成员函数(但正如本章稍后要讲到的，一些特殊成员函数——比如构造函数——不会自动继承)。我们说来自基类的这些成员被**继承**了。继承的成员函数和成员变量不在派生类的定义中列出(如后所述，有一种情况例外)，但它们会自动成为派生类的成员。例外情况是，要在派生类中更改继承的成员函数的定义，就要在派生类定义中列出它。

父类和子类

讨论派生类时可借用家庭关系术语。基类是**父类**，派生类是**子类**。这使关于继承的描述更易理解。例如，可以说子类从它的父类继承了成员变量和成员函数。比喻可更进一步。如果一个类是另一个类的父类，或者是另一个类的父类的父类……依此类推，就把它称为**前辈类**。如类 A 是类 B 的**前辈**，可以说类 B 是类 A 的**后辈**。

除了继承的成员变量和成员函数，派生类可任意添加新的成员变量和成员函数。新成员变量和新成员函数的声明在类定义中列出。例如，HourlyEmployee 类新增了两个成员变量，即 wageRate 和 hours。还新增了几个成员函数，分别是 setRate，getRate，setHours 和 getHours。图 15.4 对此进行了展示。注意，只有需要更改定义的成员函数才需列出。因此，只列出了来自基类 Employee 的成员函数 printCheck。目前不需要关心派生类的构造函数定义，下一节将讨论这个主题。

图 15.4　派生类 HourlyEmployee 的接口

```
1  // 头文件 hourlyemployee.h，它是 HourlyEmployee 类的接口
2  #ifndef HOURLYEMPLOYEE_H
3  #define HOURLYEMPLOYEE_H

4  #include <string>
5  #include "employee.h"

6  using namespace std;

7  namespace employeessavitch
8  {

9      class HourlyEmployee : public Employee
10     {
11     public:
12         HourlyEmployee();
13         HourlyEmployee(string theName, string theSSN,
14                        double theWageRate, double theHours);
15         void setRate(double newWageRate);
16         double getRate() const;
17         void setHours(double hoursWorked);
18         double getHours() const;
19         void printCheck();        <-------- 由于想更改继承的一个成员函数
20     private:                               的定义，所以这里列出了它的声明
21         double wageRate;
22         double hours;
23     };

24 } // employeessavitch

25 #endif // HOURLYMPLOYEE_H
```

图 15.5　派生类 SalariedEmployee 的接口

```
1  // 头文件 salariedemployee.h，它是 SalariedEmployee 类的接口
2  #ifndef SALARIEDEMPLOYEE_H
3  #define SALARIEDEMPLOYEE_H

4  #include <string>
5  #include "employee.h"

6  using namespace std;

7  namespace employeessavitch
8  {

9      class SalariedEmployee : public Employee
10     {
11     public:
12         SalariedEmployee();
13         SalariedEmployee (string theName, string theSSN,
14                           double theWeeklySalary);
15         double getSalary() const;
16         void setSalary(double newSalary);
17         void printCheck();
18     private:
19         double salary; // 周薪
20     };

21 }// employeessavitch

22 #endif // SALARIEDEMPLOYEE_H
```

在派生类实现文件中(例如图 15.6 的 HourlyEmployee 类的实现)，要给出所有新增成员函数的定义。注意除非继承的成员函数的定义要在派生类中改变，否则不要给出这些成

员函数的定义。

继承的成员函数的定义可在派生类的定义中修改，使其在派生类中的含义有别于在基类中的含义。这叫做对继承的成员函数进行**重定义**(redefining)。例如，派生类 HourlyEmployee 重定义了成员函数 printCheck()。为了重定义成员函数，和在派生类中新增成员函数一样，在类定义中列出它并提供新定义即可。例如，图 15.4 和图 15.6 重定义了 HourlyEmployee 类的 printCheck()函数。

图 15.6　派生类 HourlyEmployee 类的实现

```
1  // 这是 hourlyemployee.cpp 文件，它是 HourlyEmployee 类的实现
2  // HourlyEmployee 类的接口在头文件 hourlyemployee.h 中
3  #include <string>
4  #include <iostream>
5  #include "hourlyemployee.h"
6  using namespace std;

7  namespace employeessavitch
8  {
9     HourlyEmployee::HourlyEmployee() : Employee(), wageRate(0), hours(0)
10    {
11        // 主体有意留空
12    }

13    HourlyEmployee::HourlyEmployee(string theName, string theNumber,
14                                   double theWageRate, double theHours)
15    : Employee(theName, theNumber), wageRate(theWageRate), hours(theHours)
16
17    {
18        // 主体有意留空
19    }

20    void HourlyEmployee::setRate(double newWageRate)
21    {
22        wageRate = newWageRate;
23    }

24    double HourlyEmployee::getRate() const
25    {
26        return wageRate;
27    }

28    void HourlyEmployee::setHours(double hoursWorked)
29    {
30        hours = hoursWorked;
31    }

32    double HourlyEmployee::getHours() const
33    {
34        return hours;
35    }

36    void HourlyEmployee::printCheck()
37    {
38        setNetPay(hours * wageRate);
39        cout << "\n________________________________________\n";
40        cout << "Pay to the order of " << getName() << endl;
41        cout << "The sum of " << getNetPay() << " Dollars\n";
42        cout << "________________________________________\n";
43        cout << "Check Stub: NOT NEGOTIABLE\n";
44        cout << "Employee Number: " << getSSN() << endl;
45        cout << "Hourly Employee. \nHours worked: " << hours
46            << " Rate: " << wageRate << " Pay: " << getNetPay() << endl;
47        cout << "_________________________________________\n";
48    }

51 } // employeessavitch
```

选择在 printCheck 函数中设置 netPay，是因为在这里设置才有意义。注意，在派生类中重定义 printCheck 函数时，C++允许拿掉 const 修饰符

SalariedEmployee 是 Employee 类的另一个派生类。SalariedEmployee 的接口在图 15.5 中给出。将对象声明为 SalariedEmployee 类型，它将包含 Employee 的所有成员函数和成员变量。此外，还包含在 SalariedEmployee 类的定义中给出的新成员。SalariedEmployee 类没有列出任何继承的变量。相反，它只列出了来自 Employee 基类的一个函数(printCheck)，因为该函数的定义要在 SalariedEmployee 类中修改。SalariedEmployee 类最终拥有继承的三个成员变量，即 name，ssn 和 netPay；以及一个新增的成员变量 salary。注意不必列出 Employee 类的成员变量和成员函数，例如 name 和 setName。SalariedEmployee 自动拥有这些成员。

注意 Employee 类包含 HourlyEmployee 和 SalariedEmployee 这两个类通用的代码。这就避免了写两遍完全一样的代码：一遍为了 HourlyEmployee 类；另一遍为了 SalariedEmployee 类。总之，继承允许重用 Employee 类的代码。

派生类的对象具有多个类型

根据经验，我们知道计时工属于(IS-A)员工。在 C++中道理一样。由于 HourlyEmployee 是 Employee 类的派生类，所以凡是能使用 Employee 类的对象的地方都能使用 HourlyEmployee 类的对象。具体地说，如函数要求使用 Employee 类型的参数，就可为该参数传递 HourlyEmployee 类型的值。另外，可将 HourlyEmployee 类的对象赋给 Employee 类型的变量。但要注意，不可将 Employee 对象赋给 HourlyEmployee 类型的变量。毕竟，Employee 不一定是 HourlyEmployee(计时工都是员工，但员工不都是计时工)。当然，同样的道理也适用于任何基类及其派生类。在允许使用基类对象的任何地方，都能使用派生类的对象。

更笼统地说，类类型的对象可在允许使用前辈类对象的任何地方使用。假定 Child(子)类派生自 Ancestor(前辈)，而 Grandchild(孙子)类派生自 Child(子)类，那么凡是能使用 Child(子)类对象的地方，都能使用 Grandchild(孙子)类的对象。同样地，凡是能使用 Ancestor(前辈)类对象的地方，都能使用 Grandchild(孙子)类的对象。

派生类中的构造函数

基类构造函数不被派生类继承，但可在派生类构造函数的定义中调用基类构造函数，这通常正是我们需要的。派生类构造函数以特殊方式使用基类构造函数。基类构造函数会初始化从基类继承的所有数据，所以派生类构造函数最先干的活儿就是调用基类构造函数。

要用特殊语法调用基类构造函数，具体可参见图 15.6 给出的 HourlyEmployee 类的构造函数定义。为便于讨论，下面复制了 HourlyEmployee 类的一个构造函数的定义：

```
HourlyEmployee::HourlyEmployee(string theName, string theNumber,
                                double theWageRate, double theHours)
: Employee(theName, theNumber), wageRate(theWageRate), hours(theHours)
{
      // 主体有意留空
}
```

冒号后的部分是 HourlyEmployee::HourlyEmployee 构造函数定义的初始化区域。其中，Employee(theName, theNumber)调用基类 Employee 的双参数构造函数。注意，调用基类构造函数使用的语法类似于设置成员变量的语法：wageRate(theWageRate)将成员变量 wageRate 的值设置成 theWageRate；Employee(theName, theNumber)则调用基类构造函数 Employee，并传递 theName 和 theNumber 这两个参数值。由于所有工作都在初始化区域完成了，所以构造函数定义主体为空。

下面复制了图 15.6 的 HourlyEmployee 类的另一个构造函数：

```
HourlyEmployee::HourlyEmployee() : Employee(), wageRate(0), hours(0)
{
    // 主体有意留空
}
```

在这个构造函数定义中，调用基类默认(无参)构造函数来初始化继承的成员变量。在派生类构造函数的初始化区域，始终都应该包括对某个基类构造函数的调用。

如派生类构造函数定义不包括对基类构造函数的调用，会自动调用基类构造函数的默认(无参)版本。因此，在 HourlyEmployee 类的以下默认构造函数定义中，虽然省略了 Employee()，但效果与上面讨论的版本一样：

```
HourlyEmployee::HourlyEmployee() : wageRate(0), hours(0)
{
    // 主体有意留空
}
```

但最好还是显式包含对基类构造函数的调用，即使它会被自动调用。

派生类对象拥有基类的所有成员变量。调用派生类构造函数时，需要为这些成员变量分配内存，并对其进行初始化。必须由基类构造函数为继承的成员变量分配内存，基类构造函数也最适合对这些继承的成员变量进行初始化。正是由于这些原因，所以在为派生类定义构造函数时，应始终包含对基类构造函数的调用。不包含对基类构造函数的调用(在派生类构造函数定义的初始化区域)，会自动调用基类的默认(无参)构造函数(如基类没有默认构造函数就会出错)。

对基类构造函数的调用是派生类构造函数采取的第一项行动。因此，假定类 B 派生自类 A，而类 C 派生自类 B，那么在创建类 C 的对象时，会首先调用类 A 的构造函数，再调用类 B 的构造函数，最后才会执行类 C 的构造函数的剩余行动。

派生类中的构造函数

派生类不从基类继承构造函数。但为派生类定义构造函数时，可以(而且应该)包含对某个基类构造函数的调用，并将该调用放到构造函数定义的初始化区域。

不包括对任何基类构造函数的调用，在调用派生类构造函数时，会自动调用基类的默认(无参)构造函数。

陷阱：使用来自基类的私有成员变量

HourlyEmployee 类(图 15.4 和图 15.6)的对象从 Employee 类(图 15.2 和图 15.3)继承了名为 name 的成员变量。例如，以下代码将 joe 对象的成员变量 name 的值设为"Josephine"(还会将成员变量 ssn 设为"123-45-6789"，将 wageRate 和 hours 设为 0)。

```
HourlyEmployee joe("Josephine", "123-45-6789", 0, 0);
```

要把 joe.name 变成"Mighty-Joe"，可使用以下语句：

```
joe.setName("Mighty-Joe");
```

但操纵 name 这种继承的成员变量时必须小心。HourlyEmployee 类的成员变量 name 从 Employee 类继承。但成员变量 name 在 Employee 类的定义中是私有成员变量。也就是说，name 只有在 Employee 类的成员函数的定义中才可直接访问。在其他任何类(包括派生类)的成员函数定义中，不能直接通过名称访问基类的私有成员变量(或成员函数)。所以，虽然 HourlyEmployee 类有名为 name 的成员变量(从基类 Employee 继承)，但在 HourlyEmployee 类定义的任何成员函数中，对成员变量 name 的任何直接访问都是非法的。

例如，下面摘录了 HourlyEmployee::printCheck 成员函数主体的前几行(图 15.6)：

```
void HourlyEmployee::printCheck()
{
    setNetPay(hours * wageRate);

    cout << "\n________________________________________\n";
    cout << "Pay to the order of " << getName() << endl;
    cout << "The sum of " << getNetPay() << " Dollars\n";
```

你可能会奇怪，为什么要用成员函数 setNetPay 设置成员变量 netPay 的值。不小心的话会将成员函数定义的开头部分写成下面这样：

```
void HourlyEmployee::printCheck()
{                                          非法使用 netPay
    netPay = hours * wageRate;
```

正如旁注所说，上述代码非法。成员变量 netPay 是 Employee 类的私有成员变量。虽然像 HourlyEmployee 这样的派生类会继承变量 netPay，但却不能直接访问它。必须通过某个公共成员函数访问成员变量 netPay。为了完成 HourlyEmployee 类的 printCheck 的定义，正确的写法应该像图 15.6 那样(或参见前面摘录的代码)。

正是因为 name 和 netPay 等继承的变量在基类中私有，所以需要在 HourlyEmployee::printCheck 的定义中使用取值函数 getName 和 getNetPay，而不能直接使用 name 和 netPay 等变量名。不能引用继承的私有成员变量名称。相反，必须使用基类中定义的公共取值和赋值成员函数(比如 getName 和 setName)。记住，**取值函数**(accessor function)是允许访问类的成员变量的函数，而**赋值函数**(mutator function)是允许更改类的成员变量的函数。取值和赋值函数的详情已在第 10 章讨论。

许多人觉得不解，在派生类的成员函数定义中，为什么要限制对基类私有成员变量的访问？如果你是计时工，并希望更改自己的姓名，没人会说：“对不起，你改不了，你的 name 是 Employee 类私有的。”计时工不也是员工吗？这个情况在 Java 中也是一样的。在

Java 中，HourlyEmployee 类的对象也是 Employee 类的对象。但必须像刚才描述的那样，遵循私有成员变量和私有成员函数的使用法则。否则，它们的保密性会被破坏。假如类的私有成员变量可在派生类的成员函数定义中访问，那么任何时候想要访问私有成员变量，都可以创建派生类，并在那个类的成员函数中访问它。这意味着稍微花一点功夫，任何人都能访问到所有私有成员变量。故意篡改数据还不是最大的问题，最大的问题是那些不经意之间犯下的错误。假如类的私有成员变量能在派生类成员函数定义中访问，完全可能因为不慎而更改成员变量，或采取不恰当的方式更改。记住，在取值函数和赋值函数中，可防止对成员变量进行不恰当的更改。

稍后将探讨如何通过一种方式突破对基类私有成员变量的限制。 ■

陷阱：私有成员函数不会继承

如上一节所述，除非在基类的接口与实现中，否则不能直接访问基类私有成员变量(或成员函数)——即使在派生类的成员函数定义中。注意，私有成员函数和私有变量相似，两者都不能直接访问。但成员函数的限制更严格。起码还能通过取值或赋值成员函数访问私有变量，但私有成员函数根本不能使用。事实上，私有成员函数根本不会继承。

但这应该不是问题。私有成员函数只应作为辅助函数使用，所以只应在定义它们的类中使用。如成员函数在大量派生类中作为辅助成员函数使用，根本就不能算是“辅助函数”。在这种情况下，应将其设为公共成员函数。 ■

protected 限定符

如前所述，不能在派生类定义或实现中访问私有成员变量或私有成员函数。但还有一种成员变量和成员函数，可以(而且只能)在派生类中通过名称直接访问。在非派生类的其他任何类中，则不能访问这种成员。在类的成员变量或成员函数之前添加限定符 protected(而不是 private 或者 public)，那么对于除派生类之外的其他任何类或函数，它的效果等同于用 private 标记成员。在派生类中则可通过名称访问这些成员。

以派生自基类 Employee 的 HourlyEmployee 类为例。前面要求在 HourlyEmployee::printCheck 的定义中使用取值和赋值成员函数操纵继承的成员变量。如果用关键字 protected(而不是 private)标记 Employee 类的所有私有成员变量，那么在派生类 HourlyEmployee 中，HourlyEmployee::printCheck 的定义可简化成以下形式：

```
void HourlyEmployee::printCheck()
// 前提是用 protected(而不是 private)标记 Employee 的成员变量
{
    netPay = hours * wageRate;

    cout << "\n________________________________________________\n";
    cout << "Pay to the order of " << name << endl;
    cout << "The sum of " << netPay << " Dollars\n";
    cout << "_________________________________________________\n";
    cout << "Check Stub: NOT NEGOTIABLE\n";
    cout << "Employee Number: " << ssn << endl;
    cout << "Hourly Employee. \nHours worked: " << hours
         << " Rate: " << wageRate << " Pay: " << netPay << endl;
```

```
    cout << "_______________________________________\n";
}
```

在派生类 HourlyEmployee 中，继承的成员变量 name，netPay 和 ssn 可直接通过名称来访问，前提是它们已在基类 Employee 中标记成 protected(而非 private)。但在不是从 Employee 类派生的任何类中，仍然认为它们是 private 成员变量。

基类的 protected 成员变量是“受保护的”，在任何派生类中也相当于用 protected 标记。例如，假定再定义 HourlyEmployee 类的派生类 PartTimeHourlyEmployee，这个类会继承 HourlyEmployee 类的所有成员变量，包括 HourlyEmployee 从 Employee 类继承的。因此，PartTimeHourlyEmployee 类会拥有成员变量 netPay，name 和 ssn。如果这些成员变量在 Employee 类中标记为 protected，那么在 PartTimeHourlyEmployee 类的各个成员函数的定义中，可直接通过名称来访问它们。

除非在派生类(以及派生类的派生类)中，否则 protected 成员变量相当于 private 成员变量。

之所以要讨论 protected 成员变量，主要是因为你可能在其他地方看到它的使用，所以应当熟悉它们。许多(但并非所有)权威人士说，使用 protected 成员变量是一种不好的风格。他们认为这种变量违背了“隐藏类实现细节”原则。如果所有成员变量都用 private 标记，在派生类函数定义中就不能通过名称来直接访问继承的成员变量。但这并不是一件坏事，因为可调用继承的取值或赋值函数，间接读取或更改继承的 private 变量。至于是否应该使用 protected 成员，各人说法不一，所以要自行斟酌。

> **受保护成员**
>
> 在类的成员变量前使用限定符 protected(而不是 private 或 public)，对于除派生类(或派生类的派生类)之外的其他任何类或函数，它等同于使用 private 来标记。但在派生类的成员函数的定义中，可通过名称直接访问这种变量。类似地，在类的成员函数前使用限定符 protected，对于除派生类(或派生类的派生类)之外的其他任何类或函数，它等同于使用 private 来标记。但在派生类的成员函数的定义中，可使用这种 protected 函数。
>
> 派生类继承的 protected 成员仍然是 protected 的。换言之，只要在基类中将成员标记为 protected，在所有后辈类中(而非仅仅从基类直接派生的类)，都可通过成员的名称直接访问。

√ 自测题

1. 以下程序是否合法(假定已添加了合适的#include 和 using 预编译指令)?

```
void showEmployeeData(const Employee object);

int main()
{
    HourlyEmployee joe("Mighty Joe", "123-45-6789", 20.50, 40);
    SalariedEmployee boss("Mr. Big Shot", "987-65-4321", 10500.50);
    showEmployeeData(joe);
    showEmployeeData(boss);

    return 0;
```

```
}

void showEmployeeData(const Employee object)
{
    cout << "Name: " << object.getName() << endl;
    cout << "Social Security Number: "
         << object.getSSN() << endl;
}
```

2. 给出 SmartBut 类的定义，它是基类 Smart 的派生类，基类定义如下所示。暂时忽略有关#include 预编译指令和命名空间的细节。

```
class Smart
{
public:
    Smart();
    void printAnswer() const;
protected:
    int a;
    int b;
};
```

在派生类 SmartBut 中添加 bool 变量 crazy；添加一个额外的无参成员函数，返回 bool 值；再添加合适的构造函数。将新函数命名为 isCrazy。注意不需要给出任何实现，只需给出类定义。

3. 在自测题 2 讨论的派生类 SmartBut 中，成员函数 isCrazy 的以下定义合法吗？请对你的答案进行解释(注意，本题只问是否合法，不问是否合理)。

```
bool SmartBut::isCrazy() const
{
    if (a > b)
        return false;
    else
        return true;
}
```

重定义成员函数

派生类 HourlyEmployee 的定义(图 15.4)声明了新成员函数，包括 setRate，getRate，setHours 和 getHours，还列出了从 Employee 继承的 printCheck 成员函数的声明。未列出的其他继承成员函数(比如 setName 和 setSSN)会原封不动地继承，它们在 HourlyEmployee 类和 Employee 类中的定义完全相同。定义 HourlyEmployee 这样的派生类时，只有想修改定义的继承成员函数才需列出其声明。分析图 15.6 给出的 HourlyEmployee 类的实现，会发现重定义了继承的成员函数 printCheck。图 15.7 则展示了 SalariedEmployee 类为 printCheck 函数给出的新定义。注意在两个派生类中，该函数的定义各不相同。在这种情况下，我们说 printCheck 函数在各个派生类中进行了**重定义**。

图 15.7　派生类 SalariedEmployee 的实现

```
 1  // 这是 salariedemployee.cpp 文件，包含了 SalariedEmployee 类的实现
 2  // SalariedEmployee 类的接口在头文件 salariedemployee.h 中
 3  #include <iostream>
 4  #include <string>
 5  #include "salariedemployee.h"
 6  using namespace std;

 7  namespace employeessavitch
 8  {
 9     SalariedEmployee::SalariedEmployee() : Employee(), salary(0)
10     {
```

```
11          // 主体有意留空
12      }
13      SalariedEmployee::SalariedEmployee(string theName, string theNumber,
14         double theWeeklySalary)
15                     : Employee(theName, theNumber), salary(theWeeklySalary)
16      {
17          // 主体有意留空
18      }

19        double SalariedEmployee::getSalary() const
20      {
21          return salary;
22      }

23      void SalariedEmployee::setSalary(double newSalary)
24      {
25          salary = newSalary;
26      }

27      void SalariedEmployee::printCheck()
28      {
29          setNetPay(salary);
30          cout << "\n___________________________________________\n";
31          cout << "Pay to the order of " << getName() << endl;
32          cout << "The sum of " << getNetPay() << " Dollars\n";
33          cout << "_________________________________________\n";
34          cout << "Check Stub NOT NEGOTIABLE \n";
35          cout << "Employee Number: " << getSSN() << endl;
36          cout << "Salaried Employee. Regular Pay: "
37               << salary << endl;
38          cout << "_________________________________________\n";
39      }
40  } // employeessavitch
```

重定义继承的函数

派生类继承基类的所有成员函数(和成员变量)。但是，如果派生类要以不同方式实现继承的成员函数，可以在派生类中重定义该函数。要重定义成员函数，必须在派生类定义中列出它的声明，即使该声明与基类中的声明完全相同。不想重定义从基类继承的成员函数，就不要列出。

图 15.8 的示范程序演示了如何使用派生类 HourlyEmployee 和 SalariedEmployee。

图 15.8　使用派生类

```
 1  #include <iostream>
 2  #include "hourlyemployee.h"
 3  #include "salariedemployee.h"
 4  using std::cout;
 5  using std::endl;
 6  using namespace employeessavitch;

 7  int main()
 8  {
 9      HourlyEmployee joe;
10      joe.setName("Mighty Joe");
11      joe.setSSN("123-45-6789");
12      joe.setRate(20.50);
13      joe.setHours(40);
```

```
14      cout << "Check for " << joe.getName()
15           << " for " << joe.getHours() << " hours.\n";
16      joe.printCheck();
17      cout << endl;

18      SalariedEmployee boss("Mr. Big Shot", "987-65-4321", 10500.50);
19      cout << "Check for " << boss.getName() << endl;
20      boss.printCheck();
21      return 0;
22  }
```

setName，setSSN，setRate，setHours 和 getName 从 Employee 类原封不动地继承。printCheck 被重定义。getHours 函数是派生类 HourlyEmployee 新增的

示范对话

```
Check for Mighty Joe for 40 hours.
_________________________________________________
Pay to the order of Mighty Joe
The sum of 820 Dollars
_________________________________________________
Check Stub: NOT NEGOTIABLE
Employee Number: 123-45-6789
Hourly Employee.
Hours worked: 40 Rate: 20.5 Pay: 820
_________________________________________________
Check for Mr. Big Shot
_________________________________________________
Pay to the order of Mr. Big Shot
The sum of 10500.5 Dollars
_________________________________________________
Check Stub NOT NEGOTIABLE
Employee Number: 987-65-4321
Salaried Employee. Regular Pay: 10500.5
_________________________________________________
```

重定义与重载的比较

不要混淆在派生类中对函数的重定义(redefining)和对函数名的重载(overloading)。重定义时，派生类给出的新定义具有相同的参数数量和类型。另一方面，和基类的函数定义相比，如派生类的函数使用了数量不同的参数，或某个参数具有不同类型，那么派生类中实际会同时存在两个函数，这称为重载而不是重定义。例如，假定在 HourlyEmployee 类的定义中添加具有以下声明的函数：

```
void setName(string firstName, string lastName);
```

HourlyEmployee 类除了拥有这个接收两个参数的 setName 函数，还会继承以下单参数版本的 setName：

```
void setName(string newName);
```

这样，HourlyEmployee 类就有了两个名为 setName 的函数，这称为对函数名 setName 进行“重载”。

另一方面，Employee 和 HourlyEmployee 类都定义了具有以下函数声明的函数：

```
void printCheck();
```

这种情况下，HourlyEmployee 类只有一个名为 printCheck 的函数，但 HourlyEmployee

类中的 printCheck 函数定义有别于它在 Employee 类中的定义，这称为对 printCheck 函数进行了“重定义”。

混淆重定义和重载不算是一件很危险的事情，两者都合法。重要的是学会如何使用而不是区分。但无论如何，都应理解两者的差异。

> **签 名**
>
> 函数**签名**是指“函数名+参数列表中的类型序列”，不包括关键字 const 和符号&。基于这个对签名的定义，重载函数名时，函数名的两个定义必须具有不同的签名。[①]
>
> 派生类函数如果与基类函数同名，但具有不同签名，就是重载而非重定义。

访问重定义的基函数

视频讲解：*Inheritance Example*

假定重定义了函数，使其在派生类中的定义有别于基类中的定义。这种情况下，并不是说基类中的定义再也不能由派生类的对象使用了。要为派生类的一个对象调用函数的基类版本，需要用一种方式表示：“我想使用基类的函数定义(虽然我是派生类的对象)。”为此，需要使用作用域解析操作符，并指定基类名称。下面举例说明。

以图 15.2 的基类 Employee 和图 15.4 的派生类 HourlyEmployee 为例。两个类都定义了 printCheck() 函数。现在，假定每个类都有一个对象：

```
Employee janeE;
HourlyEmployee sallyH;
```

那么以下语句使用 Employee 类的 printCheck 定义：

```
janeE.printCheck();
```

而以下语句使用 HourlyEmployee 类的 printCheck 定义：

```
sallyH.printCheck();
```

在派生类对象 sallyH 上调用基类版本的 printCheck，应使用以下语句：

```
sallyH.Employee::printCheck();
```

当然，目前 Employee 类给出的那个版本的 printCheck 没太大意义。但这只是一个例子。对于其他类和其他函数，偶尔还是需要为派生类的对象使用基类的函数定义。自测题 6 展示了一个例子。

自测题

4. SalariedEmployee 类从基类 Employee 继承了 getName 和 printCheck 这两个函数(还继承了另一些东西)。但在 SalariedEmployee 类的定义中，只列出了 printCheck 函数的声明。为什么不在 SalariedEmployee 类的定义中列出 getName 函数的声明？

---

① 某些编译器的重载标准会将 const 考虑在内，而不是排除在外。但你不能依赖于这种行为，所以暂时不要考虑 const 的问题。也正是由于这个原因，一些教科书在定义“签名”时，包括了修饰符 const。这是一个见仁见智的问题。在成为专家之前，最好不要把 const 看成是“签名”的一部分。

5. 给出 TitledEmployee 类的定义，它是基类 SalariedEmployee(参见图 15.5)的派生类。TitledEmployee 类新增了一个 string 类型的成员变量 title。还新增了两个成员函数，分别是 getTitle 和 setTitle。前者无参，返回一个 string；后者是 void 函数，获取一个 string 参数。另外，TitledEmployee 类重定义了成员函数 setName。不需要给出任何实现，只需给出类定义。然而，一定要给出所有必要的#include 预编译指令以及所有 *using namespace* 预编译指令。将 TitledEmployee 类放到命名空间 employeessavitch 中。

6. 针对你在上一题给出的 TitledEmployee 类的定义，给出它的构造函数的定义。另外，给出成员函数 setName 的重定义。setName 函数应在姓名中插入职务(title)。暂不必关心#include 预编译指令和命名空间的细节。

15.2 继 承 细 节

细节决定成败。

——*俗语*

本节讨论继承的一些比较微妙的细节，大多数主题只涉及使用了动态数组或指针(以及其他动态数据)的类。

不继承的函数

假定 Derived 类是基类 Base 的派生类，Base 类的所有“普通”函数都会成为 Derived 类的继承成员。不过，某些特殊函数是不继承的。以前已见到过这样的特殊函数。例如，构造函数就不继承，私有成员函数也不继承。另外，析构函数也是不继承的。

再说拷贝构造函数，它也是不继承的，但如果在派生类(或其他任何类)中没有定义拷贝构造函数，C++会自动生成一个。但该默认拷贝构造函数只能拷贝成员变量的内容。成员变量使用了指针或动态数据，它就无法正常工作。所以，如果类的成员变量涉及指针、动态数组或其他动态数据，就应该为该类定义拷贝构造函数。无论该类是不是派生类，都应该这么做。

赋值操作符=也不继承。如基类 Base 定义了赋值操作符，但派生类 Derived 没有定义，那么 Derived 类虽然也有一个赋值操作符，但它是 C++创建的默认赋值操作符，和 Base 定义的基类赋值操作符没有任何关系。

有充分的理由不继承构造函数、析构函数和赋值操作符。为正确执行任务，它们需要获得基类无法提供的信息。为正确执行，它们必须了解派生类中引入的新的成员变量。

派生类中的赋值操作符和拷贝构造函数

重载的赋值操作符和构造函数不会继承，但它们可以(而且通常必须)在派生类的重载赋值操作符及拷贝构造函数的定义中使用。

在派生类中重载赋值操作符时，通常要使用来自基类的重载赋值操作符。下面要展示一个提纲，概括了如何写这样的代码。为理解这个代码提纲，请记住一点：重载的赋值操作符必须定义成类的成员函数。

假定 Derived 是 Base 的派生类，在 Derived 类中，重载的赋值操作符的定义通常像

下面这样开始：

```
Derived& Derived::operator =(const Derived& rightSide)
{
    Base::operator =(rightSide);
```

定义主体的第一行调用 Base 类重载的赋值操作符，这就照顾到了继承的成员变量及其数据。接着，重载的赋值操作符的定义可开始设置在 Derived 类定义中新引入的成员变量。

在派生类中定义拷贝构造函数的情况相似。假定 Derived 是 Base 的派生类，那么 Derived 类的拷贝构造函数的定义通常要用 Base 类的拷贝构造函数来设置继承的成员变量及其数据。代码通常像下面这样开始：

```
Derived::Derived(const Derived& object)
                : Base(object), <可能还有更多的初始化>
{
```

调用基类拷贝构造函数 Base(object)，可为准备创建的 Derived 对象设置继承的成员变量。注意，由于 object 是 Derived 类型，所以也是 Base 类型；因此，object 是传给 Base 类拷贝构造函数的合法实参。

当然，除非基类中已经有了一个能正确工作的赋值操作符和拷贝构造函数，否则这些都是无用功。也就是说，在基类定义中，必须包括拷贝构造函数和要么自动创建、要么重载的赋值操作符，它们在基类中必须能正确工作。

派生类中的析构函数

如基类有一个能正确工作的析构函数，在基类的派生类中，很容易定义一个能正确工作的析构函数。调用派生类的析构函数时，它会自动调用基类析构函数，所以不必显式调用；它肯定是自动发生的。因此，在派生类的析构函数中，只需用 delete 销毁派生类新增的成员变量(以及它们指向的任何数据)。与此同时，基类析构函数会负责为继承的成员变量调用 delete。

假定类 B 派生自类 A，而类 C 派生自类 B，那么一旦类 C 的某个对象离开作用域，首先会调用类 C 的析构函数，再调用类 B 的析构函数，最后调用类 A 的析构函数。注意，析构函数的调用顺序刚好与构造函数相反。

√ 自测题

7. 我们知道，重载的赋值操作符和拷贝构造函数不会继承。但这是否意味着，如果不为派生类定义重载的赋值操作符或者拷贝构造函数，派生类就没有赋值操作符和拷贝构造函数呢？

8. 假定 Child 是 Parent 的派生类，Grandchild 是 Child 的派生类。本题要考察你对这三个类(Parent、Child 和 Grandchild)的构造函数和析构函数的理解。调用 Grandchild 类的构造函数时，会以*什么*顺序调用*哪些*构造函数？调用 Grandchild 类的析构函数时，会以*什么*顺序调用*哪些*析构函数？

9. 为下面这个类给出成员函数 addValue、拷贝构造函数、重载的赋值操作符以及析构函数的定义。该类的对象用于表示一个部分填充的数组。成员变量 numberUsed 包含了目前实际填充的数组位置数。下面给的构造函数定义将帮助你解这道题。

```
#include <iostream>
#include <cstdlib>
using namespace std;
```

```
class PartFilledArray
{
public:
    PartFilledArray(int arraySize);
    PartFilledArray(const PartFilledArray& object);
    ~PartFilledArray();
    void operator = (const PartFilledArray& rightSide);
    void addValue(double newEntry);
    // 可能有更多成员函数，但它们与本题无关
protected:
    double *a;
    int maxNumber;
    int numberUsed;
};

PartFilledArray::PartFilledArray(int arraySize)
            : maxNumber(arraySize), numberUsed(0)
{
    a = new double[maxNumber];
}
```

(许多权威人士认为，成员变量应设为 private，而不要设为 protected。本书倾向于这种说法。但做作业时用一下 protected 没什么问题。另外，应该有使用 protected 变量的经验，目前确实有一些程序员在用。)

10. 定义名为 PartFilledArrayWMax 的类，它是 PartFilledArray 类的派生类。PartFilledArrayWMax 类新增了名为 maxValue 的成员变量，它容纳数组中存储的最大值。定义名为 getMax 的成员取值函数，返回数组中存储的最大值。重定义成员函数 addValue，并定义两个构造函数，其中一个获取代表数组最大元素数的 int 参数。再定义一个拷贝构造函数，一个重载的赋值操作符以及一个析构函数(真正的类有更多的成员函数，但本题不要求那么多)。

15.3 多 态 性

I did it my way. [①]

——*MY WAY BY FRANK SINATRA*

多态性(Polymorphism)是指为一个函数名关联多种含义的能力。具体地说，多态性指通过名为“晚期绑定”的特殊机制为函数名关联多个含义。多态性是面向对象编程的核心概念。本节将晚期绑定与多态性结合到一起讨论。

晚期绑定

虚函数是某种意义上能在定义前使用的函数。例如，一个图形程序可能支持几种类型的图形，比如矩形、圆、椭圆等。每种图形都可能是一个不同类的对象。假定 Rectangle(矩形)类提供了用于表示高度、宽度和中心点的成员变量，而 Circle(圆)类提供了用于表示中心点和半径的成员变量。在精心设计的编程项目中，所有这些类都可能是一个父类的后辈。假定那个父类名为 Figure(图形)。现在，假定要在屏幕上画一幅图。如画圆，就需要有别于画矩形的指令。所以，每个类都需要一个不同的函数来画它那种形状的图。但由于函数从属于类，所以它们能统一命名为 draw。假定 r 是 Rectangle 对象，c 是 Circle 对象，

① 源自 Frank Sinatra 的“My Way”这首歌，意思是“我行我素”，暗指“多态性”。——译注

那么 r.draw()和 c.draw()就是用不同代码来实现的两个函数。所有这一切都很好理解，也是我们以前学过的东西。但是，现在讨论你以前未学过的东西：在父类 Figure 中定义的虚函数。

现在，父类 Figure 可能提供了适合所有图形的函数。例如，它可能有 center 函数，能将一幅图移至屏幕中心——具体是先删除原来的图形，再在屏幕中心重画。Figure::center 可利用 draw 函数在屏幕中心重新画图。如果考虑为 Rectangle 和 Circle 类的图使用继承的 center 函数，事情就会变得棘手。

为了更清楚地理解问题，也为了使讨论更有趣，假定 Figure 类已经写好，而且已投入使用，但在以后的某个时间，我们为一种全新的图形添加了一个类，假定它就是 Triangle(三角形)类。另外，假定 Triangle 类是 Figure 类的派生类，所以它会从 Figure 类继承 center 函数。这种情况下，center 函数应该正确应用于所有 Triangle 对象。但问题来了，center 函数会使用 draw，但 draw 函数对于每种类型的图形来说都是有区别的。继承的 center 函数如果不进行任何特殊处理，就会使用 Figure 类给出的 draw 函数定义。显然，那个函数定义不能正确支持 Triangle。我们希望继承的 center 函数使用 Triangle::draw 函数，而不是使用 Figure::draw 函数。但在编写和编译 center 函数的时候(它在 Figure 类中定义)，Triangle 和 Triangle::draw 函数根本就没有写好。那么，center 函数怎样才能正确作用于 Triangle？当初编译 center 时，编译器根本不了解有关 Triangle::draw 的任何事情。假如 draw 是虚函数，一切问题都迎刃而解。

将函数指定为 virtual，相当于告诉编译器："我不知道这个函数如何实现。等它在程序中使用时，再从对象实例中获得它的实现。"在运行时确定一个过程的具体实现，这种技术称为**晚期绑定**或**动态绑定**。虚函数正是 C++提供晚期绑定的一种具体手段。但是，简单的介绍到此为止。我们需要一个例子来进一步体验(并学习怎样在程序中使用虚函数)。为解释 C++虚函数的细节，我们要用一个更简单的例子，它不会像画图程序那样复杂。

C++虚函数

假定要为一家汽车零件商店设计记账程序。你希望程序非常灵活，但不保证提前考虑到所有情况。例如，你想跟踪销售，但不能预测所有销售类型。刚开始，可能只有面向零售顾客的普通销售。这些顾客直接到店选购特定零件。但以后可能增加打折销售或要求计算运费的邮购销售。所有这些销售都是围绕具有基本价格的某种商品展开的，而且所有销售最终都会产生一个账单。对于简单的销售，账单只涉及基本价格。但如果添加了折扣，那么某些种类的账单还要依赖于折扣的幅度。程序计算每天销售总额，它是所有独立销售账单的总和。你可能还想计算一天中金额最大和最小的销售，或者一天的平均销售额。所有这些都可根据一天中生成的所有独立的账单而计算，但用于计算账单的函数要等到以后再添加，也就是在确定了要处理的销售类型之后。为适应这种情况，我们将计算账单的函数设为虚函数(为简化问题，假定每笔销售只涉及一件商品，虽然使用派生类和虚函数，完全能照顾到每次销售多件商品的情况)。

图 15.8 和图 15.19 展示了 Sale 类的接口和实现。所有类型的销售都是 Sale 类的派生类。Sale 类对应于单件商品的简单销售，无附加折扣或其他费用。注意，bill 函数的声明使用了保留字 virtual(参见图 15.9)。在图 15.10 中，注意成员函数 savings 和重载操

作符<都使用了 bill 函数。由于 bill 是虚函数，所以以后能定义 Sale 类的派生类，并为它们定义 bill 函数的不同版本。届时，Sale 类的成员函数 savings 和重载操作符<的定义将使用与派生类的对象对应的那个版本的 bill 函数。

图 15.9　基类 Sale 的接口

```
1  // 头文件 sale.h，它是 Sale 类的接口。Sale 是代表简单销售的一个类
2  #ifndef SALE_H
3  #define SALE_H
4
5  #include <iostream>
6  using namespace std;
7
8  namespace salesavitch
9  {
10
11     class Sale
12     {
13     public:
14         Sale();
15         Sale(double thePrice);
16         virtual double bill() const;
17         double savings(const Sale& other) const;
18         // 如果购买 other 代表的商品，而非购买调用对象代表的商品，返回能省多少钱
19     protected:
20         double price;
21     };
22
23     bool operator <(const Sale& first, const Sale& second);
24     // 比较两个销售，看哪个较大
25
26 } // salesavitch
27
28 #endif // SALE_H
```

图 15.10　基类 Sale 的实现

```
1  // 这是实现文件 sale.cpp，它实现了 Sale 类
2  // Sale 类的接口在头文件 sale.h 中
3  #include "sale.h"
4
5  namespace salesavitch
6  {
7
8      Sale::Sale() : price(0)
9      {}
10
11     Sale::Sale(double thePrice) : price(thePrice)
12     {}
13
14     double Sale::bill() const
15     {
16         return price;
17     }
18
19     double Sale::savings(const Sale& other) const
20     {
21         return ( bill() - other.bill() );
22     }
23
24     bool operator < (const Sale& first, const Sale& second)
25     {
26         return (first.bill() < second.bill());
27     }
28
29 } // salesavitch
```

例如，图 15.11 展示了派生类 DiscountSale(打折销售)。注意，DiscountSale 类要求为它的那个版本的 bill 函数使用不同的定义。无论如何，为 DiscountSale 类的对象使用成员函数 savings 和重载操作符<时，实际使用的是 DiscountSale 给出的那个版本的 bill 函数定义。这是 C++非常巧妙的一个设计。下面分析为 DiscountSale 类的 d1 和 d2 对象进行的函数调用 d1.savings(d2)。savings 函数的定义(即使它用于 DiscountSale 类的对象)是在基类 Sale 的实现文件中给出的。它在 DiscountSale 类出现之前就已经编译好。但在函数调用 d1.savings(d2)中，调用 bill 函数的那一行知道自己应该使用 DiscountSale 类给出的 bill 函数定义。

图 15.11 派生类 DiscountSale

这是文件 discountsale.h

```
1  // 这是 DiscountSale 类的接口
2  #ifndef DISCOUNTSALE_H
3  #define DISCOUNTSALE_H
4  #include "sale.h"
5
6  namespace salesavitch
7  {
8      class DiscountSale : public Sale
9      {
10     public:
11         DiscountSale();
12         DiscountSale(double thePrice, double theDiscount);
13         //Discount is expressed as a percent of the price.
14         virtual double bill() const;
15     protected:
16         double discount;
17     };
18 }//salesavitch
19 #endif //DISCOUNTSALE_H
```

关键字 virtual 并非必须，但作为一种良好的编码风格，最好还是将其包含在内

这是文件 discountsale.cpp

```
20 // 这是 DiscountSale 类的实现
21 #include "discountsale.h"
22
23 namespace salesavitch
24 {
25     DiscountSale::DiscountSale() : Sale(), discount(0)
26     {}
27
28     DiscountSale::DiscountSale(double thePrice, double theDiscount)
29             : Sale(thePrice), discount(theDiscount)
30     {}
31
32     double DiscountSale::bill() const
33     {
34         double fraction = discount/100;
35         return (1 - fraction)*price;
36     }
37 } // salesavitch
```

这是如何实现的？如果只是为了写 C++程序，你只需假定一切都会魔术般地发生。但如果想弄懂原理，可参考上一节的简单介绍。将函数标记为 virtual 之后，相当于告诉 C++环境：“等在程序中使用该函数时，才获得与调用对象对应的函数实现。”

图 15.12 的示范程序演示了虚函数 bill 以及使用 bill 的函数在完整程序中如何工作。

图 15.12 使用虚函数

```
1  // 该程序用于演示虚函数 bill 的使用
2  #include <iostream>
3  #include "sale.h" // 并非一定需要，但由于有 ifndef，所以包括它是很保险的
4  #include "discountsale.h"
```

```
 5  using namespace std;
 6  using namespace salesavitch;
 7
 8  int main()
 9  {
10      Sale simple(10.00);// 一件价格为$10.00 的商品
11      DiscountSale discount(11.00, 10); // 一件价格为$11.00，折扣为 10%的商品
12
13      cout.setf(ios::fixed);
14      cout.setf(ios::showpoint);
15      cout.precision(2);
16
17      if (discount < simple)
18      {
19          cout << "Discounted item is cheaper.\n";
20          cout << "Savings is $" << simple.savings(discount) << endl;
21      }
22      else
23          cout << "Discounted item is not cheaper.\n";
24
25      return 0;
26  }
```

示范对话

```
Discounted item is cheaper.
Savings is $0.10
```

要在 C++中顺利使用虚函数，需掌握如下所示的技术细节。

- 如函数在派生类中的定义有别于基类中的定义，而且你希望它成为虚函数，就要为基类的函数声明添加保留字 virtual。在派生类的函数声明中，则可以不添加 virtual。函数在基类中 virtual，在派生类中自动 virtual(但为了澄清，最好派生类中也将函数声明标记为 virtual，尽管这并非必须)。
- 保留字 virtual 在函数声明中添加，不要在函数定义中添加。
- 除非使用保留字 virtual，否则不能获得虚函数，也不能获得虚函数的任何好处。

既然虚函数如此好用，为什么不将所有成员函数都设为 virtual？这似乎只有一个理由，那就是效率。编译器和“运行时”环境要为虚函数做多得多的工作。所以，无谓地将成员函数标记为 virtual 会影响程序执行效率。

重　写

虚函数定义在派生类中发生改变时，我们说函数定义被重写。一些 C++书籍区分了重定义(redefine)和重写(override)。两者都是在派生类中更改函数定义。函数是虚函数，就称为重写。如果不是，就称为重定义。对于程序员，这种区分似乎有点无聊，因为程序员在两种情况下做的事情是一样的。不过，编译器对于这两种情况确实是区别对待的。

多　态　性

多态性是指借助晚期绑定技术，为一个函数名关联多种含义的能力。因此，多态性、晚期绑定和虚函数其实是同一个主题。

√ 自测题

11. 假定修改 Sale 类的定义(图 15.9)，删除其中的保留字 virtual，图 15.12 的输出会发生什么变化？

虚函数和扩展类型兼容性

本节讨论将类成员函数声明为 virtual 后产生的深远影响，并例示其中部分功能。

C++是强类型语言。也就是说，类型肯定会得到检查，类型不匹配(比如实参和形参类型不匹配，而且无法自动执行转换)会生成错误消息。强类型还意味着，虽然在某些得到良好定义的情况下，C++会执行自动类型转换，使你表面上能将一种类型的值赋给另一种类型的变量，但赋给变量的值通常必须与变量类型匹配。我们将 C++自动执行类型转换称为执行一次“强制”(coercion)。例如，C++允许将 char 或 int 类型的值赋给 double 类型的变量。但 C++不允许将 double 或 float 类型的值赋给任何整型(char，short，int，long)变量。

然而，“强类型检查”与面向对象编程中的“继承”概念产生了冲突。假定定义类 A 和类 B，并定义类 A 和类 B 的对象，那么并非肯定能在这些类型的对象之间赋值。例如，假定某个程序或程序单元包含以下类型声明：

```
class Pet
{
public:
    virtual void print();
    string name;
};

class Dog : public Pet  // 狗是宠物
{
public:
    virtual void print();  // 关键字 virtual 并非必须，但它有助于避免
                           // 混淆，是一种良好的编程风格
    string breed; // breed 代表狗的品种
};
Dog vDog;
Pet vPet;
```

重点注意数据成员 name 和 breed(为简化这个例子，我们将成员变量设为 public。但在实际的应用程序中应设为 private，并通过函数来操纵)。

任何 Dog(狗)都是 Pet(宠物)。所以，从表面看，Dog 类型的值似乎也应是 Pet 类型的值。换言之，以下赋值语句应该是允许的：

```
vDog.name = "Tiny"; // 狗名
vDog.breed = "Great Dane"; // 狗品种
vPet = vDog;
```

C++确实允许这种形式的赋值。可将值(比如 vDog 的值)赋给父类型的变量(比如 vPet)，反之则不成立。

虽然上述赋值是允许的，但赋给变量 vPet 的值会丢失其 breed 字段。这称为**切割问题(slicing problem)**。例如，以下输出语句会报错：

```
cout << vPet.breed;  // 非法：Pet 类没有名为 breed 的成员
```

你可能会辩称，这是有意义的，因为一旦 Dog 移动到 Pet 类型的变量，就应该像对待其他

任何 Pet 那样对待它，不应该再有 Dog 的特殊属性。然而，这已属于哲学的范畴。从编程角度说，它只会带来麻烦。名为 Tiny 的狗仍然是一条 breed(品种)为 Great Dane(大丹狗)的狗，所以我们一直都需要引用它的品种，即使它一直都是 Pet (宠物)。

幸好，C++提供了一种方式，允许在将一个 Dog 视为 Pet 的同时不丢失品种名称。为此，需要使用指向动态对象实例的指针。假定添加以下声明：

```
Pet *pPet;
Dog *pDog;
```

使用指针和动态变量，就可将 Tiny 视为 Pet 而不丢失其品种名称。以下语句是允许的：

```
pDog = new Dog;
pDog->name = "Tiny";
pDog->bred = "Great Dane";
pPet = pDog;
```

此外，仍然能访问 pPet 所指向那个节点的 breed 字段。假定像下面这样定义 Dog::print()：

```
// 使用 iostream：
void Dog::print()
{
    cout << "name: " << name << endl;
    cout << "breed: " << breed << endl;
}
```

那么以下语句：

```
pPet->print();
```

会导致在屏幕上打印以下内容：

```
name: Tiny
breed: Great Dane
```

这是由于 print()是 virtual 成员函数。我们在图 15.13 中提供了测试代码。

图 15.13　虚函数的继承

```
1  // 该程序用于演示用虚函数来解决切割问题(slicing problem)

2  #include <string>
3  #include <iostream>
4  using namespace std;
5
6  class Pet
7  {
8  public:
9      virtual void print();
10     string name;
11 };
12
13 class Dog : public Pet
14 {
15 public:
16     virtual void print();     // 关键字 virtual 并非必须，
17                               // 但它有助于避免混淆，是一种良好的编程风格
18     string breed;             // breed 代表狗的品种
19 };
20
21 int main()
22 {
```

```
23      Dog vDog;
24      Pet vPet;
25
26      vDog.name = "Tiny";
27      vDog.breed = "Great Dane";
28      vPet = vDog;
29
30      // vPet.breed;是非法的，因为 Pet 类没有名为 breed 的成员
31
32      Dog *pDog;
33      pDog = new Dog;
34      pDog->name = "Tiny";
35      pDog->breed = "Great Dane";
36
37      Pet *pPet;
38      pPet = pDog;
39      pPet->print();
40      pDog->print(); // 两个打印的是相同的输出，即 name: Tiny breed: Great Dane
42
43      // 以下语句直接访问成员变量(而不是通过虚函数)，所以会出错：
44      // cout << "name: " << pPet->name << "  breed: "
45      //        << pPet->breed << endl;
46      // 生成的错误消息是：'class Pet'没有名为'breed'的成员
47      // 欲知详情，请参见下一节
48
49      return 0;
50  }
51
52  void Dog::print()
53  {
54      cout << "name: " << name << endl;
55      cout << "breed: " << breed << endl;
56  }
57
58  void Pet::print()
59  {
60      cout << "name: " << endl; // 注意，这里并没有提到 breed
61  }
```

示范对话

```
name: Tiny
breed: Great Dane
name: Tiny
breed: Great Dane
```

陷阱：切割问题

虽然将派生类对象赋给基类变量合法，但会产生切割问题，部分数据会被“切掉”。派生类对象有、基类没有的数据成员会在赋值过程中丢失，基类没有的成员函数也会丢失。在最终的基类对象中，将无法使用这些丢失的成员。

如果进行以下声明和赋值：

```
Dog vDog;
Pet vPet;
vDog.name = "Tiny";
vDog.breed = "Great Dane";
vPet = vDog;
```

那么 vPet 不能作为 Dog 的新成员函数的调用对象，数据成员 Dog::breed 也会丢失。 ■

陷阱：不使用虚成员函数

使用 virtual 成员函数才能真正利用前面讨论的扩展类型兼容性。例如，假定不用图 15.13 的成员函数，将以下语句：

```
pPet->print();
```

替换成以下代码会报错：

```
cout << "name: " << pet->name
     << " breed: " << pPet->breed << endl;
```

这是由于对于以下表达式：

```
*pPet
```

它的类型由 pPet 的指针类型决定，它是 Pet 类的指针类型，但 Pet 类没有 breed 字段。

但是，基类 Pet 将 print()声明为 virtual。所以，一旦编译器看到以下调用就会检查 Pet 和 Dog 的 virtual 表，判断 pPet 指向的是 Dog 类型的对象：

```
pPet->print();
```

因此，它会使用为以下 print()函数生成的代码：

```
Dog::print()
```

而不是使用为以下 print()函数生成的代码：

```
Pet::print()
```

配合动态变量进行面向对象编程是一种全然不同的编程方式。刚开始，许多人都会不知所措。但只要记住以下两条简单的规则，理解起来就容易得多。

(1) 如果指针 pAncestor 的域类型是指针 pDescendant 的域类型的基类，则以下指针赋值操作是允许的：

```
pAncestor = pDescendant;
```

此外，pDescendant 指向的动态变量的任何数据成员或成员函数都不会丢失。

(2) 虽然动态变量所有附加字段(成员)都没有丢，但要用 virtual 成员函数访问。 ■

陷阱：试图对虚成员函数定义不齐全的类进行编译

增量开发是明智的开发策略。也就是说，最开始只需写少量代码，对其进行测试；再多写少量代码，对多写的代码进行测试；依此类推。然而，如编译含有 virtual 成员函数的类，但尚未实现每个成员，就可能产生极难理解的错误消息——即使没有调用未定义的成员函数！

编译前，如果还有任何尚未实现的 virtual 成员函数，编译就会失败，并产生形如“undefined reference to *Class_Name* virtual table”的错误消息。即使没有派生类，而且只有一个 virtual 成员，只要函数没有定义，就会产生这种形式的消息。

造成这种错误消息很难理解的另一个原因是，如果没有为声明为 virtual 的函数提供定义，可能还会产生进一步的错误消息，声称程序对默认构造函数进行了未定义的引用，即使你确实已定义了这些构造函数。 ■

编程提示：使析构函数成为虚函数

析构函数最好都是虚函数。但在解释它为什么好之前，首先解释一下析构函数和指针如何交互，以及虚析构函数的具体含义。

例如以下代码，其中 SomeClass 是含有非虚析构函数的类：

```
SomeClass *p = new SomeClass;
    . . .
delete p;
```

为 p 调用 delete，会自动调用 SomeClass 类的析构函数。现在看看将析构函数标记为 virtual 之后会发生什么。

为了描述析构函数与虚函数机制的交互，最简单的方式是将所有析构函数都视为同名(即使它们并非真的同名)。例如，假定 Derived 类是 Base 类的派生类，并假定 Base 类的析构函数标记为 virtual。现在分析以下代码：

```
Base *pBase = new Derived;
    . . .
delete pBase;
```

为 pBase 调用 delete 时，会调用一个析构函数。由于 Base 类中的析构函数标记为 virtual，而且指向的对象是 Derived 类型，所以会调用 Derived 的析构函数(它进而调用 Base 类的析构函数)。如果 Base 类的析构函数没有标记为 virtual，则只调用 Base 类的析构函数。

还要注意一点，将析构函数标记为 virtual 后，派生类的所有析构函数都自动成为 virtual 的(不管是否用 virtual 标记)。同样，这种行为就好比所有析构函数具有相同的名称(即使事实上不同名)。

现在，我们已准备好解释为什么所有析构函数都应该是虚函数。假定 Base 类有一个指针类型的成员变量 pB，Base 类的构造函数会创建由 pB 指向的一个动态变量，而 Base 类的析构函数会删除 pB 指向的动态变量。另外，假定 Base 类的析构函数没有标记为 virtual，并假定 Derived 类(它从 Base 派生)有一个指针类型的成员变量 pD，Derived 类的构造函数会创建由 pD 指向的一个动态变量，而 Derived 类的析构函数会删除 pD 指向的动态变量。分析以下代码：

```
Base *pBase = new Derived;
    . . .
delete pBase;
```

由于基类析构函数没有标记为 virtual，所以只会调用 Base 类的析构函数。这会将 pB 指向的动态变量所占用的内存返还给自由存储。但对于 pD 指向的动态变量，它占用的内存永远不会返还给自由存储(除非程序终止)。

另一方面，将基类 Base 的析构函数标记为 virtual，为 pBase 调用 delete 时，会调用 Derived 类的析构函数(因为指向的对象是 Derived 类型)。Derived 类的析构函数会删除 pD 指向的动态变量，再自动调用基类 Base 的析构函数。后者会删除 pB 指向的动态变量。

因此，将基类析构函数标记为 virtual，所有内存都会成功地被自由存储回收。为准备迎接这种情况，最好总是将析构函数标记为 virtual。■

√ 自测题

12. 为什么不能将基类对象赋给派生类变量？
13. 将派生类对象赋给基类变量(这是合法的)，可能产生什么问题？
14. 假定基类和派生类分别提供了一个具有相同签名的成员函数。如果你有指向某个基类对象的一个指针，并通过该指针调用一个成员函数，请解释是什么决定了最终调用的函数(可能调用基类的成员函数，或者调用派生类的成员函数)。

小 结

- 继承是代码重用机制。利用继承，可以从一个类派生出另一个类，并在派生类中添加新功能。
- 派生类会继承基类的所有成员，并可添加新成员。
- 晚期绑定是指在运行时决定使用成员函数的哪一个版本。C++利用虚函数实现晚期绑定。多态性、晚期绑定以及虚函数实际是同一主题。
- 基类 `protected` 成员可在公共派生类的成员函数中直接访问。

自测题答案

1. 合法。可为基类类型的参数传递派生类。HourlyEmployee 属于 Employee，SalariedEmployee 同样属于 Employee。

2.
```
class SmartBut : public Smart
{
public:
    SmartBut();
    SmartBut(int newA, int newB, bool newCrazy);
    bool isCrazy() const;
private:
    bool crazy;
};
```

3. 合法，因为 a 和 b 在基类 Smart 中标记为 protected，所以能在派生类中直接根据它们的名称来访问它们。相反，假如 a 和 b 标记为 private，这个定义就是非法的。

4. getName 函数的声明之所以未在 SalariedEmployee 的定义中给出，是因为它没有在 SalariedEmployee 中重定义。因此，它会从基类 Employee 原封不动地继承。

5.
```
#include <iostream>
#include "salariedemployee.h"
using namespace std;
namespace employeessavitch
{
    class TitledEmployee : public SalariedEmployee
    {
    public:
        TitledEmployee();
        TitledEmployee(string theName, string theTitle,
                       string theSSN, double theSalary);
        string getTitle() const;
        void setTitle(string theTitle);
        void setName(string theName);
    private:
        string title;
    };
} // employeessavitch
```

6.
```
namespace employeessavitch
{
    TitledEmployee::TitledEmployee()
        : SalariedEmployee(), title("No title yet")
    {
        // 主体有意留空
    }

    TitledEmployee::TitledEmployee(string theName,
                     string theTitle,
                     string theSSN, double theSalary)
        :SalariedEmployee(theName, theSSN, theSalary),
                     title(theTitle)
    {
        // 主体有意留空
    }
    void TitledEmployee::setName(string theName)
    {
        Employee::setName(title + theName);
    }
} // employeessavitch
```

7. 不是这样的。不为派生类定义重载的赋值操作符或拷贝构造函数，会为派生类自动定义一个默认赋值操作符以及一个默认拷贝构造函数。但如果类中使用了指针、动态数组或其他动态数据，那么几乎可

以肯定这些默认版本不会产生你预期的行为。

8. 构造函数的调用顺序是：首先 Parent，其次 Child，最后 Grandchild。析构函数则以相反顺序调用：首先 Grandchild，其次 Child，最后 Parent。

9.
```
// 使用 iostream 和 cstdlib:
void PartFilledArray::addValue(double newEntry)
{
     if (numberUsed == maxNumber)
     {
        cout << "Adding to a full array.\n";
        exit(1);
     }
    else
    {
        a[numberUsed] = newEntry;
        numberUsed++;
    }
}

PartFilledArray::PartFilledArray
          (const PartFilledArray& object)
               : maxNumber(object.maxNumber),
                   numberUsed(object.numberUsed)
{
    a = new double[maxNumber];

    for (int i = 0; i < numberUsed; i++)
        a[i] = object.a[i];
}

void PartFilledArray::operator =
                 (const PartFilledArray& rightSide)
{
    if (rightSide.maxNumber > maxNumber)
    {
        delete [] a;
        maxNumber = rightSide.maxNumber;
        a = new double[maxNumber];
    }
    numberUsed = rightSide.numberUsed;

    for (int i = 0; i < numberUsed; i++)
        a[i] = rightSide.a[i];
}

PartFilledArray::~PartFilledArray()
{
    delete [] a;
}
```

10.
```
class PartFilledArrayWMax : public PartFilledArray
{
public:
    PartFilledArrayWMax(int arraySize);
    PartFilledArrayWMax(const PartFilledArrayWMax& object);
    ~PartFilledArrayWMax();
    void operator= (const PartFilledArrayWMax& rightSide);
    void addValue(double newEntry);
    double getMax();
private:
    double maxValue;
};

PartFilledArrayWMax::PartFilledArrayWMax(int arraySize)
                        : PartFilledArray(arraySize)
{
    // 主体有意留空
    // MaxValue 未初始化，因为没有合适的默认值
}
```

```
/*
注意，以下语句不能工作，因为它调用 PartFilledArray 的
默认构造函数，但 PartFilledArray 没有默认构造函数：
PartFilledArrayWMax::PartFilledArrayWMax(int arraySize)
               : maxNumber(arraySize), numberUsed(0)
{
    a = new double[maxNumber];
}
*/

PartFilledArrayWMax::PartFilledArrayWMax
            (const PartFilledArrayWMax& object)
                : PartFilledArray(object)
{
    if (object.numberUsed > 0)
    {
        maxValue = a[0];

        for (int i = 1; i < numberUsed; i++)
            if (a[i] > maxValue)
                maxValue = a[i];
    }// 否则保持 maxValue 的未初始化状态
}

// 这等价于 C++提供的默认析构函数，所以能省略这个定义
// 但是，如果省略它，必须同时在类的定义中省略析构函数的声明

PartFilledArrayWMax::~PartFilledArrayWMax()
{
    // 主体有意留空
}

void PartFilledArrayWMax::operator =
                 (const PartFilledArrayWMax& rightSide)
{
    PartFilledArray::operator = (rightSide);
    maxValue = rightSide.maxValue;
}

// 使用 iostream 和 cstdlib:
void PartFilledArrayWMax::addValue(double newEntry)
{
   if (numberUsed == maxNumber)
   {
       cout << "Adding to a full array.\n";
       exit(1);
   }

    if ((numberUsed == 0) || (newEntry > maxValue))
        maxValue = newEntry;

    a[numberUsed] = newEntry;
    numberUsed++;
}

double PartFilledArrayWMax::getMax()
{
    return maxValue;
}
```

11. 输出会变成下面这样：

```
Discounted item is not cheaper.
```

12. 因为无法为派生类新增的成员赋值。

13. 虽然将派生类对象赋给基类变量是合法操作，但在此期间，派生类对象中不属于基类成员的那些部分会被丢弃。这称为“切割问题”(slicing problem)。

14. 如基类函数有 virtual 修饰符，就由“初始化指针的对象”的类型决定调用谁的成员函数。如基类成员函数没有 virtual 修饰符，就由“指针”的类型决定调用谁的成员函数。

编 程 练 习

编程练习一般只需写很小的程序，运用本章提到的编程概念。

1. 写程序使用如图 15.5 所示的 SalariedEmployee 类。程序要定义名为 Administrator(行政人员)的类，它从 SalariedEmployee 类派生。可在基类中将 private 变成 protected。要求提供以下新增的数据和函数成员。
   - 一个 string 类型的成员变量，包含行政人员职务，比如 Director(总监)或 Vice President(副总裁)。
   - 一个 string 类型的成员函数，包含该行政人员要负责的公司职能部门，比如 Production(生产部)、Accounting(财务部)或者 Personnel(人事部)。
   - 一个 string 类型的成员函数，包含该行政人员的直接上司的姓名。
   - 一个 double 类型的 protected:成员变量，容纳该行政人员的年薪。如根据前面的建议进行了修改(将 private 变成 protected)，可考虑利用现有的 salary 成员。
   - 一个名为 setSupervisor 的成员函数，用于更改上司姓名。
   - 一个成员函数，从键盘读入行政人员的数据。
   - 一个名为 print 的成员函数，将对象数据输出到屏幕。
   - 对成员函数 printCheck()的一个重载，在支票上打印合适的内容。
2. 以图 15.2、图 15.4 和图 15.5 的层次结构为基础，添加更多的员工类别，比如 Temporary(临时工)、Administrative(行政人员)、Permanent(固定员工)等。实现并测试这个层次结构。测试所有成员函数。含有一个菜单的用户界面会为你的测试程序增色不少。
3. 视频讲解：*Solution to Practice Program 15.3*

   下面列出使用了继承的两个类的定义、实现和 main 函数。将代码放到合适的文件中，添加必要的 include 语句和预处理器语句，使程序能编译和运行。应输出“Circle has radius 2 and area 12.5664”。

```
class Shape
{
public:
    Shape();
    Shape(string name);
    string getName();
    void setName(string newName);
    virtual double getArea() = 0;
private:
    string name;
};
Shape::Shape()
{
    name="";
}
Shape::Shape(string name)
{
    this->name = name;
}
string Shape::getName()
{
    return this->name;
```

```
}
void Shape::setName(string newName)
{
    this->name = newName;
}

class Circle : public Shape
{
public:
    Circle();
    Circle(int theRadius);
    void setRadius(int newRadius);
    double getRadius();
    virtual double getArea();
private:
    int radius;
};
Circle::Circle() : Shape("Circle"), radius(0)
{ }
Circle::Circle(int theRadius) : Shape("Circle"),
               radius(theRadius)
{ }
void Circle::setRadius(int newRadius)
{
    this->radius = newRadius;
}
double Circle::getRadius()
{
    return radius;
}
double Circle::getArea()
{
    return 3.14159 * radius * radius;
}
int main()
{
    Circle c(2);
    cout << c.getName() << " has radius " <<
    c.getRadius() << " and area " <<
    c.getArea() << endl;
    return 0;
}
```

再添加一个 Rectangle 类，也从 Shape 派生。修改 Rectangle 类使它包含私有 width 和 height 变量，用于设置宽度和高度的构造函数，用于接收宽度和高度的函数，以及用于计算矩形面积的 getArea 函数。在 main 中添加以下代码，应输出“Rectangle has width 3 has height 4 and area 12.0”。

```
Rectangle r(3,4);
cout <<   r.getName() << " has width " <<
          r.getWidth() << " has height " <<
          r.getHeight() << " and area " <<
          r.getArea() << endl;
```

编 程 项 目

编程项目要求综合运用多方面的知识来解决问题，程序一般比编程练习大，解题方式多样化。

视频讲解：*Solution to Programming Project 15.1*

1. 给出 Doctor(医生)类的定义，共对象代表某家诊所的医生。该类是 SalariedEmployee 类(图 15.5)的派生类。每个 Doctor 对象都包含该医生的专业说明(specialty)，比如"Pediatrician"(儿科医生)、

"Obstetrician"(产科医生)以及"General Practitioner"(全科医生)等，所以请将 specialty 设为 string 类型。此外，Doctor 对象还包含 office visit fee(门诊费)，它应该具有 double 类型。确保你的类合理地补充了构造函数、取值成员函数、赋值成员函数、一个重载的赋值操作符以及一个拷贝构造函数。写测试程序来测试所有函数。

2. 创建名为 Vehicle(车)的基类，它含有制造商名称(string 类型)、发动机缸数(int 类型)以及车主(Person 类型，在下面给出)。然后，创建 Vehicle 的派生类 Truck(卡车)，它具有一些附加属性，其中包括以吨为单位的载重(double 类型，因为可能含有小数，比如 4.5 吨)以及以磅为单位的牵引力(int 类型)。确保你的类合理地补充了构造函数、取值成员函数、赋值成员函数、一个重载的赋值操作符以及一个拷贝构造函数。写测试程序来测试所有成员函数。

   Person 类的定义如下所示。该类的实现也是你在本项目的任务之一。

```
class Person
{
public:
    Person();
    Person(string theName);
    Person(const Person& theObject);
    string getName() const;
    Person& operator = (const Person& rtSide);
    friend istream& operator >>(istream& inStream,
                      Person& personObject);
    friend ostream& operator <<(ostream& outStream,
                     const Person& personObject);
private:
    string name;
};
```

3. 定义 Car 类，从编程项目 2 的 Vehicle 类派生。定义 SportCar 类，从 Car 类派生。选择成员变量和函数时一定要合理。写个 driver 来测试 Car 类和 SportCar 类。(我不是故意说双关语。[①])

4. 给出 Patient(患者)和 Billing(收费)这两个类的定义，其对象用于表示一家诊所的记录。Patient 派生自编程项目 2 的 Person 类。每个 Patient 对象都包含该患者的姓名(从 Person 类继承)以及主治医师(具有编程项目 2 定义的 Doctor 类型)。每个 Billing 对象都包含一个 Patient 对象、一个 Doctor 对象以及 double 类型的应付款金额。要确保你的类合理地补充了构造函数、取值成员函数、赋值成员函数、一个重载的赋值操作符以及一个拷贝构造函数。首先写测试程序来测试所有成员函数，然后写测试程序创建至少 2 个患者、至少 2 个医生以及至少 2 个 Billing 对象，打印所有 Billing 对象的总收入。

5. 假定某个图形系统提供了代表各种图形的类，包括矩形、正方形、三角形、圆等。例如，矩形可能包括高度、宽度和中心点这几个数据成员，而正方形只有中心点和边长这两个数据成员，圆只有中心点和半径这两个数据成员。在精心设计的系统中，它们可能都从一个通用的 Figure 类派生。你要实现这样的一个系统。

   Figure 类是基类。应该只添加从 Figure 派生的 Rectangle 和 Triangle 类。每个类都定义了原型成员函数 erase 和 draw。每个成员函数都输出一条消息，指出调用的是什么函数，以及调用对象属于哪个类。由于只是原型函数，所以它们只需输出消息就可以了。成员函数 center 首先调用 erase 函数删除图形，再调用 draw 函数在中心点重画图形。由于你的 erase 和 draw 函数只是原型函数，所以 center 并不能真的执行"居中"操作，它只是简单地调用 erase 和 draw 这两个成员函数。另外，在成员函数 center 中添加一条输出消息，宣布 center 已经调用。成员函数不接收任何参数。

   这个项目要分以下三部分来完成。
   a. 不用虚函数而写出类定义。编译并测试。
   b. 使基类的成员函数为 virtual。编译并测试。
   c. 解释结果为什么有区别。

   在一个真正的项目中，必须修改上述每个成员函数的代码，让它执行真正的绘图操作，编程项目 6 将

① driver 是指驱动程序，也有司机的意思。——译注

要求你这样做。用以下 main 函数执行所有测试：

```
// 该程序用于测试编程项目 5
#include <iostream>
#include "figure.h"
#include "rectangle.h"
#include "triangle.h"
using std::cout;

int main()
{
    Triangle tri;
    tri.draw();
    cout << "\nDerived class Triangle object calling center().\n";
    tri.center(); // 调用 draw 和 center

    Rectangle rect;
    rect.draw();
    cout << "\nDerived class Rectangle object calling center().\n";
    rect.center(); // 调用 draw 和 center
    return 0;
}
```

6. 对编程项目 5 进行扩展。为各个构造函数以及成员函数 Figure::center，Figure::draw，Figure::erase，Triangle::draw，Triangle::erase，Rectangle::draw 和 Rectangle::erase 提供新定义，使 draw 函数能在屏幕的合适位置显示字符'*'，从而真正实现绘图。对于 erase 函数，只需清除屏幕(通过输出空行或者执行其他更复杂的操作)。本项目涉及大量细节，所以必须自行决定其中的部分细节。

7. 银行允许开设多种类型的账户，而且针对取款这样的交易，一般都制定了不同的手续费规则。顾客可以支付一定的费用从账户取款，并将款项转账到另一个账户。

   写程序为银行账户定义基类，并定义两个派生类(如后所述)，用它们代表具有不同取款规则的账户。再写一个函数，负责将资金从一个账户(任意类型)转账到另一个。注意，“转账”的意思是从一个账户取款，然后向另一个账户存款。由于转账可能在任何时候针对任何类型的账户进行，所以类的转账函数必须是 virtual 的。写主程序创建三个账户(每个类一个)，然后测试转账函数。

   具体地说，需要创建一个名为 BankAccount 的基类，它的数据成员包括账户所有人姓名(一个字符串，假定命名为 name)，以及账户余额(一个 double 值，假定命名为 balance)。在类中添加成员函数 deposit 和 withdraw(分别获取一个 double 类型的 amount 参数)，并添加取值函数 getName 和 getBalance。deposit(存款)函数将 amount 加到 balance 上(假定 amount 非负)，而 withdraw(取款)将 amount 从 balance 中减去(假定 amount 非负，而且小于或等于 balance)。另外，还要创建一个名为 MoneyMarketAccount 的类，它从 BankAccount 派生。在一个 MoneyMarketAccount 中，用户可在给定的优惠时间内享受两次免手续费取款(暂不必关心具体有多长优惠时间)。免手续费取款达到指定次数之后，每次取款都从余额中扣除 1.50 美元的手续费。所以，该类必须有一个数据成员跟踪取款次数。另外，还必须重写 withdraw 定义。最后创建一个 CDAccount 类(用于建模“定期存款单”，即 Certificate of Deposit)，它同样从 BankAccount 派生。该类除了包含数据成员 name 和 balance 之外，还有一个名为 interestRate(利率)的数据成员。银行的规定是假如提前取款，顾客须缴纳一定的提前取款罚金(prepayment penalty)。

   假定一次取款操作(任意金额)会扣除账户年利息的 25%作为罚金。另外，假定取款金额和罚金都从账户余额中扣除。同样地，withdraw 函数必须重写基类的版本。对于所有这三个类，withdraw 函数都应返回一个表示账户状态的整数(允许取款；或者余额不足，不允许取款)。考虑到本练习的目的，暂不要关心这些账户的其他函数和属性(比如何时和怎样支付利息)。

8. 射频标识(Radio Frequency Identification，RFID)芯片是附加在其他产品上的一种小芯片。它们就像无线电条形码，可用无线方式将标识号发送给接收器。RFID 芯片用途之一是物流货运。假定有一个装满货物的集装箱。如果没有 RFID 芯片，就只能以人工方式清点集装箱内的所有货物。但是，为集装箱加装 RFID 芯片后，RFID 芯片就可以将货物清单以电子方式传送给一个人，不需要人工清点。

   为了建模这个应用，请写一个名为 ShippingContainer 的基类。它有一个整数形式的集装箱 ID 号。

有一些成员函数用于设置和访问 ID 号。添加虚函数 getManifest 返回一个空字符串。该函数的作用是返回集装箱的货物清单。

创建名为 ManualShippingContainer 的派生类，它代表以人工方式对集装箱进行清点。在这个方法中，只是简单地附加了对集装箱所有货物的一段文字描述。例如，这段文字描述可能是“4 箱苹果，10 箱梨”。添加 string 变量来存储清单。添加名为 setManifest 的函数来设置该字符串。重写 getManifest 函数，使它返回这个字符串。

创建第二个派生类 RFIDShippingContainer，代表以 RFID 方式对集装箱进行清点。为了模拟 RFID 芯片所做的事情，创建 add 函数模拟将一件货物添加到集装箱。在类中，使用你选择的数据结构存储已添加的货物及其数量的一个清单(货物名称和数量合并成一个字符串，比如"1 箱梨")。例如，假定 add 函数像下面这样调用三次：

```
rfidContainer.add("1 箱梨");      // 添加一箱梨
rfidContainer.add("1 箱苹果");    // 添加一箱苹果
rfidContainer.add("1 箱梨");      // 添加一箱梨
```

在这个时候，数据结构应存储包含两种货物的一个清单：苹果和梨。苹果数量是 1，梨的数量是 2。重写 getManifest 函数，遍历货物清单，构造包含了所有货物的一个字符串。在上例中，返回的字符串应为"2 箱梨，1 箱苹果"。

最后，写 main 程序创建一个指针数组，其中的指针要指向 6 个 ShippingContainer 对象。用 3 个 ManualShippingContainer 对象和 3 个 RFIDShippingCopntainer 对象来实例化该数组。对于 ManualShippingContainer 对象，必须调用 setManifiest 来设置货物。对于 RFIDShippingContainer 对象，必须调用 add 来设置货物(不过，假如我们说的都是真的，那么集装箱的货物应通过 RFID 芯片自行添加，不需要人工输入)。最后写循环来遍历所有 ShippingContainer 指针，并输出每个集装箱对象的货物清单以及集装箱的 ID。这个输出是集装箱的收货方希望看到的。

要将整数转换成字符串，C++11 提供的简单方式是：string s = to_string(intVariable)。

9. 这个项目的目标是创建简单的二维“捕食者 - 被捕食者”模拟。在这个模拟中，被捕食者是蚂蚁(ant)，捕食者是狮蚁(doodlebug)。这些小生物生活在 20×20 的网格中。每个单元格每次只能由一个个体占据。整个网格是封闭的，个体不允许离开世界边缘。时间以 time step 为单位。个体在每个 time step 里面都要采取某项行动。

   蚂蚁的行为像下面这样建模。

   - Move(移动)：在每个 time step 中，都随机向上、向下、向左或者向右移动。假如所选方向上的邻居单元格被占据，或者会造成蚂蚁移动到网格的边缘之外，那么蚂蚁就停留在当前单元格中。
   - Breed(繁殖)：如果一只蚂蚁在 3 个 time step 中保持存活，在第 3 个 time step 结束之后(也就是在移动之后)，该蚂蚁会繁殖。为了模拟繁殖，需要在相邻(上、下、左或者右)的一个空单元格中创建一只新蚂蚁。没有可用的空单元格，就不会繁殖。一旦成功繁殖出后代，除非再次经历 3 个 time step，否则不能繁殖另一个后代。

   狮蚁的行为像下面这样建模。

   - Move(移动)：在每个 time step 中，假如有一只相邻的蚂蚁(上、下、左或者右)，就移动到那个单元格，吃掉蚂蚁。否则，狮蚁就按照和蚂蚁相同的规则移动。注意，狮蚁不能吃掉狮蚁。
   - Breed(繁殖)：假如一只狮蚁在 8 个 time step 中保持存活，在第 8 个 time step 结束之后，会按照与蚂蚁相同的方式繁殖出一只新狮蚁。
   - Starve(饥饿)：假如一只狮蚁在连续 3 个 time step 中没有吃掉一只蚂蚁，在第三个 time step 结束之后，它会感觉饥饿并死亡。该狮蚁应从网格中拿掉。

   在一轮中，所有狮蚁都应先于蚂蚁移动。

   写程序来实现这个模拟，使用 ASCII 字符"o"表示蚂蚁，"x"表示狮蚁。创建名为 Organism(有机生物)的类，它封装了通用于蚂蚁和狮蚁的基本数据。该类应该有一个名为 move 的 virtual 函数，它要在

派生类 Ant 和 Doodlebug 中进行具体定义。可能需要额外的数据结构来跟踪已移动的生物。

使用 5 只狮蚁和 100 只蚂蚁初始化这个世界。在每个 time step 后，都提示用户按 Enter 键移动到下一个 time step。应该看到狮蚁和蚂蚁数量的循环变化。一些随机性的混乱可能造成一种或两种生物毁灭，对此要有心理准备。

10. 视频讲解：*Solution to Programming Project 15.10*

下面列出了玩猜数游戏的代码。你的任务是用代表玩家或电脑的对象来扩展这个程序。注意，使用 rand()函数需包含 cstdlib(参见附录 4)：

```
bool checkForWin(int guess, int answer)
{
  cout << "You guessed" << guess << ".";
  if (answer == guess)
  {
    cout << "You're right! You win!" << endl;
    return true;
  }
  else if (answer < guess)
      cout << "Your guess is too high." << endl;
  else
      cout << "Your guess is too low." << endl;
  return false;
}
void play(Player &player1, Player &player2)
{
  int answer = 0, guess = 0;
  answer = rand() % 100;
  bool win = false;
  while (!win)
  {
    cout << "Player 1's turn to guess." << endl;
    guess = player1.getGuess();
    win = checkForWin(guess, answer);
    if (win ) return;
    cout << "Player 2's turn to guess." << endl;
    guess = player2.getGuess();
    win = checkForWin(guess, answer);
  }
}
```

play 函数获取的输入是两个 Player 对象。为 Player 类定义名为 getGuess()的虚函数。Player::getGuess()的实现可以直接返回 0。接着，定义从 Player 派生的 HumanPlayer(玩家)类。HumanPlayer::getGuess 的实现应提示用户输入一个数字，并返回从键盘输入的值。接着，定义从 Player 派生的 ComputerPlayer(电脑)类。ComputerPlayer::getGuess()的实现应随机选择 0~99 的数(请参见第 4 章了解关于随机数生成的信息)。最后，构造 main 函数来调用 play(Player &player1, Player &player2)。第一次传递 HumanPlayer 的两个实例(相当于两个玩家在玩)，第二次传递一个 HumanPlayer 实例和一个 ComputerPlayer 实例(人机对战)，第三次传递 ComputerPlayer 的两个实例(电脑跟电脑对战)。

11. 编程项目 10 的电脑对手在玩猜数游戏时很“菜”，因为它每次只会随机猜一个数。请修改程序，使电脑有根据地进行猜测。具体采用什么战术取决于你，但必须为 Player 和 ComputerPlayer 类添加函数(可能不止一个)，使 play(Player &player1, Player &player2)函数能将猜数结果发送给电脑对手。换言之，电脑必须知道它上一次猜的数是过大还是过小。另外，它必须知道对手上一次猜的数是过大还是过小。然后，电脑可利用这些信息修正自己的下一次猜测。另外，添加必要的函数允许电脑连续游戏。

12. 以 13.2 节的 Queue 类为基础，修改它存储整数而不是字符。一种特殊类型的队列是优先级队列。优先级队列的行为和普通队列一样，只是 remove 函数总是提取具有最小值的项(也就是具有最高优先级的项)。创建从 Queue 类派生的 PriorityQueue 类，提供合适的构造函数。重定义 PriorityQueue 类中的 remove 函数，提取具有最小值的项。在 PriorityQueue 对象中添加几个数字来测试 PriorityQueue 类。删除每个数，打印删除的数。

13. 以下代码试图创建一个宠物类：

```
class Pet
{
public:
   Pet();
   void printDescription();

   string name;
   int type;
   bool neuterSpayed;
   bool talks;
};

Pet::Pet() : type(0), neuterSpayed(false),
             talks(false)
{ }

void Pet::printDescription()
{
   switch (type)
   {
      case 0:
         cout << "Dog named " << name << endl;
         cout << "Neuter/Spayed: " <<
                 neuterSpayed;
         break;
      case 1:
         cout << "Cat named " << name << endl;
         cout << "Neuter/Spayed: " <<
                 neuterSpayed;
         break;
      case 2:
         cout << "Bird named " << name << endl;
         cout << "Talks: " << talks << endl;
         break;
   }
      cout << endl;
}
```

利用继承重写代码。Pet 是基类，子类包括 Dog，Cat 和 Bird。变量应和恰当的类关联，如有必要定义成私有，并提供恰当的函数来访问变量。printDescription 重写为虚函数。switch 语句和 type 变量都应该不需要了。写 main 函数创建宠物向量或数组，其中至少包括一只鸟、一只狗和一只猫。遍历并输出每个宠物的描述。

第 16 章　异常处理

例外是对规则的证明。[①]

——*俗语(或者说“例外测试规则”)*

概述

写程序的一种方式是先假定不会出现异乎寻常或不正确的结果。例如，假定程序要从列表获取项目，就先假定列表非空。在程序能正确支持正常情况之后，再添加代码处理异常情况。C++提供了一种机制在代码中反映该思路。大致地说，先假装不会产生任何异常，写好处理正常情况的代码。之后，利用 C++的异常处理机制，添加处理异常情况的代码。

虽然异常处理通常用于处理“出错情况”，但最好将它视为“异常情况”处理机制。这是逻辑问题——能被正确处理的“错误”就不是错误。

或许，异常最重要的用途就是处理可能会遇到特殊情况的函数。特殊情况需特殊对待，具体由函数的使用方式决定。函数可能在多个程序中使用，有的程序以一种方式处理特殊情况，有的程序则以另一种方式处理。例如，假定函数遇到了“除以 0”的特殊情况，在某些程序中，程序会终止；但在另一些程序中，则可能发生其他事情。以后会知道，可定义函数在发生特殊情况时抛出异常，而那个异常允许在函数外部对特殊情况进行处理。这样在不同程序中调用该函数时，就可采取不同方式处理特殊情况。

在 C++中，异常处理的过程是：某个库软件或者你的代码提供了一种机制，能在出现异常情况时发出信号。这称为**抛出异常**(throwing the exception)。在程序另一个地方，需添加合适的代码处理异常情况，这称为**处理异常**(handling the exception)。这种编程方式可生成更有条理的代码。当然，需要学习具体如何用 C++做好这些事情。

预备知识

除了“陷阱：派生类中的异常规范”小节，整个 16.1 节只使用了第 2 章～第 6 章、第 10 章和第 11 章的内容。“陷阱”小节使用了第 15 章的内容。该小节可跳过，不妨碍知识的连贯性。

除了“测试可用内存”小节，整个 16.2 节只使用了来自第 2 章～第 8 章、第 15 章的 15.1 节以及本章 16.1 节的内容。“测试可用内存”小节使用了第 15 章的内容。该小节可跳过，不妨碍知识的连贯性。

16.1 异常处理基础

程序大多数情况下都能正常运行。我不晓得它在那种情况下也要正常运行。

——*计算机系某学生对成绩提出的申诉*

使用异常处理需谨慎。下面给出一些简单的例子，目的是帮你理解异常处理。刚开始的例子都很简单。如此简单的情况其实不必大费周章使用异常处理。

① “例外”和“异常”的英语均是 exception。——译注

异常处理的简单例子

在这个例子中，假定牛奶在我们的文化中是一种重要食物，几乎永远用不完。但是，仍然希望程序处理“用完牛奶”这种不太可能发生的情况。首先假定牛奶用不完，然后用以下基本代码处理这种正常情况：

```
cout << "Enter number of donuts:\n";          // 输入甜甜圈的数量
cin >> donuts;
cout << "Enter number of glasses of milk:\n"; // 输入牛奶的杯数
cin >> milk;
dpg = donuts/static_cast<double>(milk);       // dpg 表示每杯牛奶可以
                                              //  配多少个甜甜圈
cout << donuts << " donuts.\n"
     << milk << " glasses of milk.\n"
     << "You have " << dpg
     << " donuts for each glass of milk.\n";
```

如果没有牛奶，上述代码会产生除以 0 的情况，这属于错误。为了处理“用完牛奶”这一特殊情况，可以测试这种异常。添加了测试的完整程序在图 16.1 中给出。注意，图 16.1 的程序没有使用异常处理机制。接着，再来看如何使用 C++异常处理机制重写这个程序。

图 16.1　不借助异常处理来处理特殊情况

```
1  #include <iostream>
2  using namespace std;

3  int main()
4  {
5      int donuts, milk;
6      double dpg;
7      cout << "Enter number of donuts:\n";
8      cin >> donuts;
9      cout << "Enter number of glasses of milk:\n";
10     cin >> milk;

11     if (milk <= 0)
12     {
13         cout << donuts << " donuts, and No Milk!\n"
14              << "Go buy some milk.\n";
15     }
16     else
17     {
18         dpg = donuts/static_cast<double>(milk);
19         cout << donuts << " donuts.\n"
20              << milk << " glasses of milk.\n"
21              << "You have " << dpg
22              << " donuts for each glass of milk.\n";
23     }

24     cout << "End of program.\n";
25     return 0;
26 }
```

示范对话

```
Enter number of donuts:
12
Enter number of glasses of milk:
0
12 donuts, and No Milk!
Go buy some milk.
End of program.
```

图 16.2 重写了图 16.1 的程序，在其中使用了异常。这只是一个简单的例子。

图 16.2　使用异常处理

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      int donuts, milk;
7      double dpg;
8
9      try
10     {
11         cout << "Enter number of donuts:\n";
12         cin >> donuts;
13         cout << "Enter number of glasses of milk:\n";
14         cin >> milk;
15
16         if (milk <= 0)
17                 throw donuts;
18
19         dpg = donuts/static_cast<double>(milk);
20         cout << donuts << " donuts.\n"
21              << milk << " glasses of milk.\n"
22              << "You have " << dpg
23              << " donuts for each glass of milk.\n";
24     }
25     catch(int e)
26     {
27         cout << e << " donuts, and No Milk!\n"
28              << "Go buy some milk.\n";
29     }
30
31     cout << "End of program.\n";
32     return 0;
33 }
```

示范对话 1

```
Enter number of donuts:
12
Enter number of glasses of milk:
6
12 donuts.
6 glasses of milk.
You have 2 donuts for each glass of milk.
```

示范对话 2

```
Enter number of donuts:
12
Enter number of glasses of milk:
0
12 donuts, and No Milk!
Go buy some milk.
End of program.
```

如此简单的情况实际用不着异常，但它确实很好地演示了异常处理。虽然程序整体并没有变得更简单，但至少单词 try 和 catch 之间的代码变得更清晰了，这其实已暗示了异常的好处。仔细观察单词 try 和 catch 之间的代码，它们与图 16.1 的代码大体相同，但舍弃了巨大的 if-else 语句，换成下面这个小得多的 if 语句(以及一些简单的非分支语句)：

```
if (milk <= 0)
    throw donuts;
```

该 if 语句指出，没有牛奶就做一些特殊的事情。具体做什么由单词 catch 之后的代码指定。程序基本思路是：正常情况由单词 try 之后的代码处理，只有在异常情况下，才使用单词 catch 之后的代码。这样就成功隔离了正常情况和异常情况。在这个简单例子中，像这样的隔离并不能带来太大益处，但在其他较复杂的环境中，就显得非常有用。下面分析细节。

在 C++中处理异常时，要采取一个 try-throw-catch 三段式过程。try 块的语法如下：

```
try
{
    Some_Code // 一些代码
}
```

这个 try 块包含程序的基本算法，告诉计算机在一切正常的情况下进行什么处理。之所以叫 try 块，是因为不能百分之百保证一切正常，但你想“试一下”。

现在，假如真的出现了异常，你就想抛出这个异常。“抛出异常”是指出“一件事情不对劲”的方式。添加 throw 之后，try 块的基本提纲就变成以下形式：

```
try
{
    Code_To_Try // 要尝试的代码
    Possibly_throw_An_Exception // 可能抛出异常
    More_Code // 更多的代码
}
```

下面是包括 throw 语句的一个 try 块的例子(摘自图 16.2)：

```
try
{
        cout << "Enter number of donuts:\n";
        cin >> donuts;
        cout << "Enter number of glasses of milk:\n";
        cin >> milk;

        if (milk <= 0)
              throw donuts;

        dpg = donuts/static_cast<double>(milk);
        cout << donuts << " donuts.\n"
             << milk << " glasses of milk.\n"
             << "You have " << dpg
             << " donuts for each glass of milk.\n";
}
```

以下语句抛出 int 值 donuts：

```
throw donuts;
```

有时将抛出的值(本例是 donuts)直接称为**异常**(exception)，所以执行 throw 语句就称为**抛出异常**(throwing an exception)。可以抛出任意类型的值，本例抛出 int 值。

throw 语句

语法

```
throw Expression_for_Value_to_Be_Thrown;
```

示例

```
if (milk <= 0)
    throw donuts;
```

> 执行 throw 语句时，该语句所在的 try 块会停止执行。如 try 块之后跟有一个合适的 catch 块，控制权就转交给那个 catch 块。throw 语句几乎肯定要嵌入一个分支语句(比如 if 语句)中。可以抛出任意类型的值。

正如 throw 这个名称所暗示的，某样东西被“抛出”后，会从一个地方转移到另一个地方。在 C++中，从一个地方转移到另一个地方的是“控制权”(以及抛出的值)。抛出异常后，throw 语句所在的 try 块会停止执行，并开始执行 catch 块的代码。执行 catch 块的过程称为捕捉异常或处理异常。异常被抛出后，它最终要由某个 catch 块处理(捕捉)。在图 16.2 中，try 块之后紧接着一个合适的 catch 块。下面复制了这个 catch 块：

```
catch(int e)
{
    cout << e << " donuts, and No Milk!\n"
         << "Go buy some milk.\n";
}
```

该 catch 块看起来很像函数定义，它接收 int 类型的参数。虽然本质上不是函数定义，但从某些方面来看，catch 块确实像一个函数。它本质上是在程序遇到(并执行)以下语句之后，单独执行的一个代码块：

```
throw Some_int; // 该语句要放在一个 try 块中
```

所以，虽然该 throw 语句类似于函数调用，但并不是调用函数，而是调用 catch 块，并执行 catch 块的代码。通常将 catch 块称为**异常处理程序**[①]，暗示其风格与函数相似。

那么在 catch 块的下面这一行语句中，标识符 e 是什么？

```
catch(int e)
```

标识符 e 看起来像是参数，行为也非常接近于参数。所以，将这个 e 称为 **catch 块参数**(但要记住，这并不表示 catch 块是函数)。catch 块参数要做下面两件事情。

(1) catch 块参数之前要附加类型名，指出 catch 块能捕捉什么类型的抛出值。

(2) catch 块参数为捕捉的抛出值指定了名称，所以在 catch 块中写代码时，可对捕捉的抛出值进行处理。

下面按相反顺序讨论 catch 块参数的这两种功能。本节讨论如何将 catch 块参数作为抛出和捕捉的值的名称来使用。下一节要讨论具体由哪个 catch 块(哪个异常处理程序)处理抛出的值。目前的例子只有一个 catch 块。e 是很常用的 catch 块参数名称，但可用任何合法的标识符替代 e。

下面分析图 16.2 的 catch 块具体如何工作。一个值被抛出之后，将立即停止执行 try 块中的代码，并将控制权传递给 try 块之后的 catch 块。以下代码摘自图 16.2：

```
catch(int e)
{
    cout << e << " donuts, and No Milk!\n"
         << "Go buy some milk.\n";
```

① exception handler 按约定俗成的方式翻译成“异常处理程序”，但如你所知，它并不是程序，而是一个代码块。其他可接受的说法包括“异常处理器”和“异常处理块”等。——译注

```
}
```

一个值被抛出之后，除非该值具有 int 类型，否则无法由上述 catch 块进行处理。在图 16.2 中，抛出的值由变量 donuts 给出。由于 donuts 具有 int 类型，所以上述 catch 块能成功捕捉抛出的值。

假定 donuts 的值是 12，而 milk 的值是 0(参见图 16.2 的示范对话 2)。由于 milk 的值不为正数，所以执行 if 语句中的 throw 语句。在这种情况下，会抛出 donuts 变量的值。图 16.2 的 catch 块捕捉到 donuts 的值之后，会将这个值传给 catch 块参数 e，并执行 catch 块的代码，最终生成以下输出：

```
12 donuts, and No Milk!
Go buy some milk.
```

catch 块参数

catch 块参数是 catch 块头部中的标识符，是可能抛出的异常(一个值)的占位符。一旦前面的 try 块抛出一个(合适的)值，该值就会传给 catch 块参数。catch 块参数可为任何合法(非保留字)标识符。

示例

```
catch(int e)
{
    cout << e << " donuts, and No Milk!\n"
         << "Go buy some milk.\n";
}
```

e 就是 catch 块参数。

如 donuts 的值是正数，则不执行 throw 语句。这种情况会执行整个 try 块。执行了 try 块最后一个语句后，接着执行 catch 块之后的语句。注意，没有抛出异常，catch 块会被忽略。

从表面看，try-throw-catch 三段式过程类似于一个 if-else 语句。两者几乎完全等价，只是前者会抛出值。try-throw-catch 过程在 if-else 语句的基础上增加了向某个分支发送消息的能力。虽然从表面看，和 if-else 语句的这个区别不是很明显，但在实际应用中，它意义重大。

正式总结一下：try 块包含 throw 语句，throw 语句通常只在异常情况下执行。一旦执行，就抛出某种类型的值。抛出异常(比如图 16.2 的 donuts 值)后，try 块停止执行。try 块剩余的所有代码都被忽略，控制权移交给合适的 catch 块。catch 块与前面的 try 块对应。一旦抛出异常，异常对象就传给 catch 块参数，开始执行 catch 块中的语句。例如，观察图 16.2 的示范对话会发现一点：只要用户输入的不是正数，try 块就会停止，转而执行 catch 块。目前假定每个 try 块的后面都跟有一个合适的 catch 块。以后会讨论在没有合适 catch 块的情况下会发生什么。

再总结一下 try 块没有抛出异常会发生什么。如 try 块没有异常(值)被抛出，那么在 try 块正常结束后，程序从 catch 块之后的代码继续执行。换言之，如果没有抛出异常，catch 块会被忽略。程序执行的大多数时间，throw 语句都不会执行，所以 try 块中的代码基本上都能顺利执行完毕，并完全忽略 catch 块中的代码。

try-throw-catch

try-throw-catch 是抛出和捕捉异常的基本机制。throw 语句抛出异常(一个值)，catch 块捕捉异常(那个值)。抛出异常后，try 块终止，转而执行 catch 块的代码。catch 块结束后，继续执行 catch 块之后的代码(前提是 catch 块没有终止程序或执行另一些特殊操作)。

如 try 块没有抛出异常，在 try 块结束后，程序从 catch 块之后的代码继续执行。换言之，如果没有抛出异常，catch 块会被忽略。

语法

```
try
{
    Some Statements // 一些语句
    <要么是含有 throw 语句的代码，要么是可能抛出异常的函数调用>
    Some More Statements // 更多的语句
}
catch(Type Name e)
{
    <如 try 块抛出 catch 块参数类型的值，就执行这里的代码>
}
```

示例　参见图 16.2。

√ 自测题

1. 以下代码会输出什么？

```
int waitTime = 46;
try
{
    cout << "Try block entered.\n";
    if (waitTime > 30)
        throw waitTime;
    cout << "Leaving try block.\n";
}

catch(int thrownValue)
{
    cout << "Exception thrown with\n"
         << "waitTime equal to " << thrownValue << endl;
}
cout << "After catch block." << endl;
```

2. 在自测题 1 的代码中，如果将下面这一行：

```
int waitTime = 46;
```

替换成：

```
int waitTime = 12;
```

会产生什么输出？

3. 从自测题 1 的代码中摘抄出 throw 语句。

4. 执行一个 throw 语句后会发生什么？请笼统解释一下，不要解释自测题 1 或其他示范代码中具体发生的事情。

5. 从自测题 1 的代码中摘抄出 try 块。
6. 从自测题 1 的代码中摘抄出 catch 块。
7. 在自测题 1 的代码中，catch 块参数是什么。

定义自己的异常类

throw 语句能抛出任意类型的一个值。程序员常做的事情就是定义一个类，它的对象专门容纳抛给 catch 块的信息。之所以要定义具体的异常类，更重要的原因是可用不同类型标识每种异常情况。

异常类本质上还是类。之所以是异常类，是因为它的使用方式比较特别。不过，确定异常类名称和其他细节时最好多注意一下。图 16.3 是含有程序员自定义异常类的示范程序。它仍是一个简单程序，演示了 C++异常处理的细节。对于如此简单的任务，它显得有点儿“大材小用”，但确实很好地演示了 C++的一些细节。

注意它的 throw 语句，如下所示：

```
throw NoMilk(donuts);
```

NoMilk(donuts) 调用 NoMilk 类的构造函数。构造函数获取一个 int 参数(本例是 donuts)，创建 NoMilk 类的对象，然后“抛出”该对象。

图 16.3 定义自己的异常类

```
1  #include <iostream>
2  using namespace std;

3  class NoMilk
4  {
5  public:
6      NoMilk();
7      NoMilk(int howMany);
8      int getDonuts();
9  private:
10     int count;
11 };

12 int main()
13 {
14     int donuts, milk;
15     double dpg;

16     try
17     {
18         cout << "Enter number of donuts:\n";
19         cin >> donuts;
20         cout << "Enter number of glasses of milk:\n";
21         cin >> milk;
22         if (milk <= 0)
23                 throw NoMilk(donuts);

24         dpg = donuts/static_cast<double>(milk);
25         cout << donuts << " donuts.\n"
26              << milk << " glasses of milk.\n"
27              << "You have " << dpg
28              << " donuts for each glass of milk.\n";
29     }
30     catch(NoMilk e)
31     {
32         cout << e.getDonuts() << " donuts, and No Milk!\n"
33              << "Go buy some milk.\n";
34     }
35     cout << "End of program.";
36     return 0;
```

这是教你熟悉 C++语法的简单程序。不要将其视为异常处理的典型例子

```
37  }
38
39  NoMilk::NoMilk()
40  {}
41  NoMilk::NoMilk(int howMany) : count(howMany)
42  {}
43
44  int NoMilk::getDonuts()
45  {
46      return count;
47  }
```

示范对话与图 16.2 一样

多个 throw 块和 catch 块

try 块允许抛出任意数量的异常值，这些值可为任意类型。但在实际应用中，每次执行 try 块都只抛出一个异常(因为一旦执行 throw 就终止 try 块)。但执行 try 块时，完全可以在不同情况下抛出不同类型的异常值。每个 catch 块只能捕捉一种类型的值，但通过在一个 try 块之后添加多个 catch 块，就能捕捉不同类型的异常值。例如，图 16.4 的程序在 try 块之后添加了两个 catch 块。

注意，用于 DivideByZero 的 catch 块是无参的。如果不需要参数，只需要列出类型，不必列出参数。稍后的“编程提示：异常类可能很简单”小节会进一步讨论该问题。

图 16.4 捕捉多个异常

尽管这里没有，但异常类可以有其自己的接口和实现文件，并且可放入一个命名空间中。这又是一个简单程序

```
1   #include <iostream>
2   #include <string>
3   using namespace std;
4
5   class NegativeNumber
6   {
7   public:
8       NegativeNumber();
9       NegativeNumber(string takeMeToYourCatchBlock);
10      string getMessage();
11  private:
12      string message;
13  };
14
15  class DivideByZero
16  {};
17
18  int main()
19  {
20      int jemHadar, klingons;
21      double portion;
22
23      try
24      {
25          cout << "Enter number of Jem Hadar warriors:\n";
26          cin >> jemHadar;
27          if (jemHadar < 0)
28              throw NegativeNumber("Jem Hadar");
29
30          cout << "How many Klingon warriors do you have?\n";
31          cin >> klingons;
32          if (klingons < 0)
33              throw NegativeNumber("Klingons");
34          if (klingons != 0)
```

```
35              portion = jemHadar/static_cast<double>(klingons);
36          else
37              throw DivideByZero();
38          cout << "Each Klingon must fight "
39               << portion << " Jem Hadar.\n";
40      }
41      catch(NegativeNumber e)
42      {
43         cout << "Cannot have a negative number of "
44              << e.getMessage() << endl;
45      }
46      catch(DivideByZero)
47      {
48         cout << "Send for help.\n";
49      }
50
51      cout << "End of program.\n";
52      return 0;
53  }
54
55
56  NegativeNumber::NegativeNumber()
57  {}
58
59  NegativeNumber::NegativeNumber(string takeMeToYourCatchBlock)
60      : message(takeMeToYourCatchBlock)
61  {}
62
63  string NegativeNumber::getMessage()
64  {
65      return message;
66  }
```

示范对话 1

```
Enter number of JemHadar warriors:
1000
How many Klingon warriors do you have?
500
Each Klingon must fight 2.0 JemHadar.
End of program
```

示范对话 2

```
Enter number of JemHadar warriors:
-10
Cannot have a negative number of JemHadar
End of program.
```

示范对话 3

```
Enter number of JemHadar warriors:
1000
How many Klingon warriors do you have?
0
Send for help.
End of program.
```

陷阱：首先捕捉较具体的异常

捕捉多个异常时，catch 块的顺序可能十分重要。try 块抛出异常值后，会依次尝试后续 catch 块，与所抛异常的类型匹配的第一个 catch 块得以执行。

例如，以下是 catch 块的一个特例，它能捕捉抛出的任意类型的值：

```
catch (...)
{
   <这里随便放什么代码>
}
```

三个点并不是说要省略什么。相反，要在程序中实际输入这三个点。这是一个很好的默认 catch 块，应该把它放到其他所有 catch 块之后。例如在图 16.4 中，可把它添加到现有 catch 块之后：

```
catch(NegativeNumber e)
{
   cout << "Cannot have a negative number of "
        << e.getMessage() << endl;
}
catch(DivideByZero)
{
   cout << "Send for help.\n";
}
catch(...)
{
    cout << "Unexplained exception.\n";
}
```

但要注意，它只适合放在一组 catch 块的末尾。假定打乱这个顺序，换成下面这样：

```
catch(NegativeNumber e)
{
   cout << "Cannot have a negative number of "
        << e.getMessage() << endl;
}
catch(...)
{
    cout << "Unexplained exception.\n";
}
catch(DivideByZero)
{
   cout << "Send for help.\n";
}
```

采用这个顺序，NegativeNumber 类型的异常(抛出的值)会顺利地被 NegativeNumber catch 块捕捉。但如果抛出 DivideByZero 类型的值，会被以 catch(...)开头的 catch 块捕捉。所以，DivideByZero catch 块永远不会执行。幸好，像这样的低级错误，大多数编译器都会友善地提醒你。■

编程提示：异常类可能很简单

下面复制了图 16.4 的 DivideByZero 异常类的定义：

```
class DivideByZero
{};
```

该异常类没有成员变量和成员函数(默认构造函数除外)。虽然只有一个名称，但它非常有用。抛出 DivideByZero 类的对象，会激活相应的 catch 块，就像图 16.4 那样。

使用如此简单的异常类时，通常不对抛给 catch 块的异常(也就是抛出的值)进行任何处理。这种异常只用于将控制权转交给 catch 块。所以，可省略 catch 块参数(任何时候只要确定用不着 catch 块参数，就可将其省略，无论异常类型是否简单)。■

在函数中抛出异常

有时希望推迟处理异常。例如在某个函数中，如发现除以 0 就抛出异常。但可能不想在该函数中捕捉异常。抛出异常后，使用该函数的一些程序可能要直接终止，另一些程序则可能要做其他事情。因此，在函数内部捕捉异常，根本不知道应该拿异常怎么办。在这种情况下，不要在函数定义中捕捉异常，而是让使用该函数的程序(或其他代码)将函数调用放到 try 块中，并在 try 块之后的 catch 块中捕捉异常。

下面分析图 16.5 的程序。它有一个 try 块，但 try 块中没有任何可见的 throw 语句。在那个程序中，执行抛出操作的语句如下：

```
if (bottom == 0)
    throw DivideByZero();
```

该语句不是在 try 块中写的。但当程序执行时，该语句会在 try 块中出现，因为该语句在 safeDivide 函数中定义，而对 safeDivide 函数的调用在 try 块中进行。

图 16.5　在函数内部抛出异常

```
1  #include <iostream>
2  #include <cstdlib>
3  using namespace std;
4
5  class DivideByZero
6  {};
7
8  double safeDivide(int top, int bottom) throw (DivideByZero);
9
10 int main()
11 {
12     int numerator;
13     int denominator;
14     double quotient;
15     cout << "Enter numerator:\n";   // 输入分子
16     cin >> numerator;
17     cout << "Enter denominator:\n";  // 输入分母
18     cin >> denominator;
19
20     try
21     {
22        quotient = safeDivide(numerator, denominator);
23     }
24     catch(DivideByZero)
25     {
26         cout << "Error: Division by zero!\n"
27              << "Program aborting.\n";
28         exit(0);
29     }
30
31     cout << numerator << "/" << denominator
```

```
32          << " = " << quotient << endl;
33
34     cout << "End of program.\n";
35     return 0;
36 }
37
38
39 double safeDivide(int top, int bottom) throw (DivideByZero)
40 {
41     if (bottom == 0)
42         throw DivideByZero();
43
44     return top/static_cast<double>(bottom);
45 }
```

示范对话 1

```
Enter numerator:
5
Enter denominator:
10
5/10 = 0.5
End of Program.
```

示范对话 2

```
Enter numerator:
5
Enter denominator:
0
Error: Division by zero!
Program aborting.
```

异常规范

如函数不捕捉异常，它至少应该提醒程序员：对函数的任何调用都有可能抛出异常。如存在可能抛出的异常，但函数定义未捕捉，应该在**异常规范(exception specificaftion)**中列出那些异常类型。图 16.5 的函数声明展示了一个异常规范：

```
double safeDivide(int top, int bottom) throw (DivideByZero);
```

如图 16.5 所示，异常规范要同时放在函数声明和函数定义中。如函数有多个函数声明，所有函数声明都必须有完全一致的异常规范。函数的异常规范有时也称为 **throw 列表**。

如函数定义可能抛出多个异常，就用逗号分隔不同异常类，如下所示：

```
void someFunction() throw (DivideByZero, OtherException);
```

异常规范列出的所有异常类型都会得到正常处理。所谓“正常处理”，是指要像本节前面描述的那样处理。具体地说，要将函数调用放到 try 块中，再在 try 块之后跟随 catch 块来捕捉那种类型的异常。一旦函数抛出异常(而且未在函数内部捕捉)，则 try 块之后的 catch 块会捕捉异常。如果没有任何异常规范(throw 列表)，就连一个空白的异常规范都没有，效果等同于在异常规范中自动列出所有可能的异常类型；换言之，抛出的任何异常都会得到正常处理。

那么，在函数中抛出异常，但异常规范没有列出该异常(也没有在函数内部捕捉)，会发生什么事情？这种情况下程序会终止。具体地说，如异常在函数中抛出，但既没有在异常规范中列出，也没有在函数内部捕捉，那么它不会被任何 catch 块捕捉，而是直接导致

程序终止。这种异常称为**未处理异常**。记住，如果完全没有异常规范列表，就连空白的都没有，那么效果等同于在规范列表中列出所有异常。这种情况下抛出异常不会终止程序。

注意，异常规范是为那些要在函数外部捕捉的异常准备的。不在函数外部捕捉，就不在异常规范中列出。要在函数外部捕捉，无论起源于何处，都应在异常规范中列出。如函数定义内部的 try 块抛出异常，并在函数定义内部的 catch 块中捕捉该异常，该异常的类型就不需要在异常规范中列出。如函数定义包括对另一个函数的调用，而另一个函数可能抛出它自己不会捕捉的异常，就应该在异常规范中列出异常类型。

要表明函数抛出的任何异常都会在函数内部捕捉，没有要在外部捕捉的异常，可使用一个空白异常规范，如下所示：

```
void someFunction() throw();
```

几种情况总结如下：

```
void someFunction() throw(DivideByZero, OtherException);
// DivideByZero 或 OtherException 类型的异常会被正常处理
// 至于其他任何异常，如抛出后未在函数主体中捕捉，就终止程序

void someFunction() throw();
// 空异常列表：如抛出任何未在函数主体中捕捉的异常，就终止程序

void someFunction();
// 正常处理所有类型的所有异常
```

注意，派生类[①]的对象也是它的基类的对象。所以，如类 D 是类 B 的派生类，而类 B 在异常规范中，那么类 D 的对象在抛出之后，会得到正常处理，因为它也是类 B 的对象，而类 B 在异常规范中。然而，不会进行任何自动类型转换。例如，假定 double 在异常规范中，那么它不能处理抛出 int 值的情况。这时需要在异常规范中同时列出 int 和 double。

最后提醒一点：并非所有编译器都能像前面描述的那样处理异常规范。有些编译器实际会将异常规范视为注释。对于那些编译器，异常规范对代码不起作用。考虑到这个原因，更有必要将函数可能抛出的所有异常都放到异常规范中。这样所有编译器都会以相同方式对待你的异常(称为“编译器一致性”)。当然，也可以完全不包括任何异常规范，同样获得编译器一致性。但这样一来，你的程序就没得到良好的编档，且无法享受支持异常规范的编译器所提供的错误检查服务。如编译器确实支持异常规范，程序一旦抛出你没有预料到的异常，就会立即终止(注意，这是运行时行为，但具体的运行时行为取决于编译器)。

陷阱：派生类中的异常规范

在派生类中重定义(redefine)或重写(override)函数定义时，它应具有和基类一样的异常规范，或至少应该在新的异常规范中给出基类异常规范的一个子集。换言之，重定义或重写函数定义时，不可在异常规范中添加新异常。但如果愿意，可删减基类原有的异常。之所以有这个要求，是因为在能使用基类对象的任何地方，都能使用派生类对象。因此，重定义或重写的函数必须兼容为基类对象编写的任何代码。■

① 如果还没有完成派生类的学习，可暂时忽略有关派生类的内容。

√ 自测题

8. 以下程序输出什么？

```
#include <iostream>
using namespace std;
void sampleFunction(double test) throw (int);
int main()
{
    try
    {
       cout << "Trying.\n";
       sampleFunction(98.6);
       cout << "Trying after call.\n";
    }
    catch(int)
    {
       cout << "Catching.\n";
    }
    cout << "End of program.\n";
    return 0;
}
void sampleFunction(double test) throw (int)
{
    cout << "Starting sampleFunction.\n";
    if (test < 100)
       throw 42;
}
```

9. 在自测题 8 的程序中，如果将 try 块中的以下代码：

```
sampleFunction(98.6);
```

修改成以下形式：

```
sampleFunction(212);
```

输出有什么变化？

16.2 用于异常处理的编程技术

在特殊情况下使用。①

——*沃伦·皮斯(Warren Peace)，《中尉的武器》*

迄今为止，已通过大量代码解释了 C++异常处理机制的基本知识。但还没有任何一个例子能很好地、真正地发挥异常处理的强大能力。理解了异常处理的基本知识之后，下面深入解释异常处理技术。

抛出异常的时机

以前用非常简单的代码展示了异常处理的基本概念。但那些例子过于简单。一种更复杂、但效果更好的方案是用不同函数分隔“抛出异常”和“捕捉异常”这两种操作。大多数时候，应将任何 throw 语句放到函数定义内部，在那个函数的异常规范中列出异常，再

① 特殊情况 = 异常情况。——译注

在不同的函数中添加 catch 子句。所以，try-throw-catch 三段式操作的最佳结构如下：

```
void functionA() throw (MyException)
{
    ...
    throw MyException(<可能有一个参数>);
    ...
}
```

然后，在另一个函数中(甚至可能是其他文件中的其他函数)，使用以下形式：

```
void functionB()
{
        ...
    try
    {
        ...
        functionA();
        ...
    }
    catch(MyException e)
    {
        <处理异常>
    }
...
}
```

除此之外，即使像这样使用 throw 语句也应谨慎。除非万不得已，否则不要使用。能以其他方式轻松解决问题，就不要抛出异常。只有在需要根据函数调用方式和调用地点来决定处理异常情况的方式时，才应使用 throw 语句。如果对异常情况的处理方式要依赖于函数的调用方式与调用地点，最好让调用函数的程序员处理异常。其他任何情况都应避免抛出异常。

何时抛出异常

通常，throw 语句应在函数中使用，并为该函数列出异常规范。此外，只有在需要根据函数的调用方式和调用地点决定处理异常情况的方式时，才应使用 throw 语句。如果对异常情况的处理方式要依赖于函数的调用方式和调用地点，最好让调用函数的程序员处理异常。其他任何情况都应避免抛出异常。

陷阱：未捕捉的异常

代码抛出的每个异常都应该在代码的某个地方捕捉。如果异常在抛出之后，在任何地方都没有捕捉它，程序就会终止。■

陷阱：嵌套 try-catch 块

可将 try 块及其后续各个 catch 块放到更大的 try 块中，或放到更大的 catch 块中。个别情况下，这样做很有用。但在考虑这样做时，请想好是否有一种更好的方式组织程序。在几乎任何情况下，一种更好的做法都是将内层的 try-catch 块放到一个函数定义中，再

将对该函数的调用放到外层的try块或catch块中(有时甚至能完全删除一个或多个多余的try块)。

如果将 try 块及其后续 catch 块放到更大的 try 块中，而且内层 try 块抛出但未捕捉一个异常，则异常会被抛到外层 try 块，以便在那里处理，甚至在那里捕捉。■

陷阱：滥用异常

滥用异常，程序将产生复杂的控制流程，最终导致程序完全无法理解。更糟的是，只要稍不留意，异常就很容易被滥用。通过抛出异常，可将控制权从程序的一个地方转移到程序中的其他几乎任何地方。在编程的早期岁月，这种不加限制的控制权转移是通过称为 goto 的结构来实现的。但编程专家早就达成共识，像这种不加限制的控制权转移是非常不好的编程风格。异常使你回到那个年代。使用异常需谨慎，而且只以特定方式使用。一个不错的规则是：只要产生了写 throw 语句的念头，就赶快想一想如果没有这个 throw 语句，是不是也能写出程序或类定义。能以其他方式写代码，就放弃 throw 语句。■

异常类层次结构

视频讲解：*The STL Exception Class*

定义异常类的层次结构可能很有用。例如，可先定义 ArithmeticError(数学错误)异常类，再定义 DivideByZeroError(除以 0 错误)异常类，将其作为 ArithmeticError 类的派生类。由于 DivideByZeroError 属于 ArithmeticError，所以为 ArithmeticError 设计的每个 catch 块都能顺利捕捉 DivideByZeroError。在异常规范中列出 ArithmeticError，实际也在异常规范中列出了 DivideByZeroError——虽然 DivideByZeroError 这一名称未在异常规范中出现。

测试可用内存

第 13 章曾用以下代码新建动态变量：

```
struct Node
{
    int data;
    Node *link;
};
typedef Node* NodePtr;
   . . .
NodePtr pointer = new Node;
```

只要有足够内存创建新节点，上述代码就能正确执行。但内存不足会发生什么？如果没有足够的内存创建节点，会抛出 bad_alloc 异常。bad_alloc 类型是 C++语言的一部分，是定义好的，你不用定义它。

由于 new 在内存不足时抛出 bad_alloc 异常，所以可像下面这样检查是否内存耗尽：

```
try
{
    NodePtr pointer = new Node;
}
```

```
catch (bad_alloc)
{
    cout << "Ran out of memory!";
}
```

当然，除了给出警告消息，还能做其他许多事情，但具体细节取决于当前的编程项目。

重新抛出异常

在 catch 块中抛出异常是合法的。极少数情况下，可能希望捕捉一个异常，再根据当前实际情况抛出同一个异常或抛出一个不同的异常，以便在异常处理链中进一步处理。

√ 自测题

10. 如果一个异常没有在任何地方捕捉，会发生什么情况？
11. 可在 try 块中嵌套另一个 try 块吗？

小　结

- 异常处理允许为正常情况和异常情况分别编码。
- 异常可在 try 块中抛出。另外，也可在不含 try 块的函数定义中抛出异常。那个函数甚至不必用 catch 块捕捉该异常。可在位于程序其他地方的 try 块中插入对该函数的调用。
- 异常在 catch 块中捕捉。
- try 块后面可以跟随多个 catch 块。这种情况下，首先要列出比较具体的异常类进行处理的 catch 块，再列出对较一般的异常类进行处理的 catch 块。
- 异常能不用就尽量不用，切忌滥用。

自测题答案

1.
```
Try block entered.
Exception thrown with
waitTime equal to 46
After catch block.
```

2.
```
Try block entered.
Leaving try block.
After catch block.
```

3.
```
throw waitTime;
```

注意以下语句是 if 语句，而不是 throw 语句，虽然其中包含了一个 throw 语句：

```
if (waitTime > 30)
    throw waitTime;
```

4. 执行 throw 语句时，它所在的 try 块会终止，try 块剩余的所有语句被忽略，控制权转交给后续 catch 块。转交控制权时，还会将抛出的值传给 catch 块参数(如果有的话)，随即执行 catch 块的代码。

5.
```
try
{
    cout << "Try block entered.\n";
    if (waitTime > 30)
        throw waitTime;
    cout << "Leaving try block.\n";
}
```

6.
```
catch(int thrownValue)
{
    cout << "Exception thrown with\n"
         << "waitTime equal to " << thrownValue << endl;
}
```

7. thrownValue 就是 catch 块参数。

8.
```
Trying.
Starting sampleFunction.
Catching.
End of program.
```

9.
```
Trying.
Starting sampleFunction.
Trying after call.
End of program.
```

10. 如异常在任何地方都未被捕捉，程序终止。

11. 可以，可将 try 块及其相应的 catch 块放到更大的 try 块中。但更好的做法是将内层 try 块和 catch 块放到函数定义中，再在外层 try 块中调用那个函数。

编 程 练 习

编程练习一般只需写很小的程序，运用本章提到的编程概念。

1. 视频讲解：*Solution to Practice Program 16.1*

返回特殊错误码的函数最好改为抛出异常。这样可避免忽略错误代码，或者错将其当作有效数据。下面这个类维护账户余额：

```
class Account
```

```
{
private:
    double balance;
public:
    Account()
    {
        balance = 0;
    }
    Account(double initialDeposit)
    {
        balance = initialDeposit;
    }
    double getBalance()
    {
        return balance;
    }

    // 存款函数。返回新余额。如有错，返回-1
    double deposit(double amount)
    {
        if (amount > 0)
            balance += amount;
        else
            return -1; // 该代码表示有错
        return balance;
    }

    // 取款函数。返回新余额。如金额无效，返回-1
    double withdraw(double amount)
    {
        if ((amount > balance) || (amount < 0))
            return -1;
        else
            balance -= amount;
        return balance;
    }
};
```

重写这个类，抛出恰当的异常，而不是返回-1 作为错误码。写测试代码尝试存取无效金额，捕捉抛出的异常。

2. 标准模板库包含一个 exception 类，它是 STL 函数抛出的所有异常的基类。所以捕捉它可捕捉任何异常。以下代码为 STL 异常设置一个 try-catch 块：

```
#include <iostream>
#include <string>
#include <exception>
using namespace std;

int main()
{
    string s = "hello";
    try
    {
        cout << "No exception thrown." << endl;
    }
    catch (exception& e)
    {
        cout  << "Exception caught: "
              << e.what() << endl;
    }
    return 0;
}
```

修改代码在 try 块中抛出异常。可尝试使用 at 成员函数访问字符串中的无效索引以抛出异常。

编 程 项 目

编程项目要求综合运用多方面的知识来解决问题，程序一般比编程练习大，解题方式多样化。

1. 写程序将 24 小时制时间换算为 12 小时制。以下是一个示范对话：

```
Enter time in 24-hour notation:
13:07
That is the same as
1:07 PM
Again?(y/n)
y
Enter time in 24-hour notation:
10:15
That is the same as
10:15 AM
Again?(y/n)
y
Enter time in 24-hour notation:
10:65
There is no such time as 10:65
Try again:
Enter time in 24-hour notation:
16:05
That is the same as
4:05 PM
Again?(y/n)
n
End of program
```

   要定义一个名为 TimeFormatMistake 的异常类。如用户输入非法时间，比如 10:65(甚至一些垃圾字符，比如&&*68，程序应抛出并捕捉一个 TimeFormatMistake 异常)。

2. 写程序将日期从数字月/日格式换算为字母月/日格式，比如 1/31 或 01/31 换算为 January 31。对话与编程项目 1 相似。要定义两个异常类：一个叫 MonthError；另一个叫 DayError。如果输入的不是有效月份(1～12 的整数)，程序抛出并捕捉一个 MonthError。类似地，如果输入的不是有效天数(1～29、30 或 31 的整数，具体取决于月份)，则程序抛出并捕捉一个 DayError。为简化问题，始终允许 2 月有 29 天(换言之，避免涉及闰年的计算)。

3. 视频讲解：*Solution to Programming Project 16.3*

   写程序输入 1～10 的值，输出文本柱形图，将每个值的出现次数换算成*的个数。程序首先询问用户准备输入多少个数。随后，假如输入的值不是全数字，或超过 1～10 的范围，就捕捉一个异常。

   提示：将每个数作为字符串来输入。扫描字符串，检查它是否全是数字，否则抛出异常。使用以下代码将字符串 str 转换成整数：

```
atoi(str.c_str());
```

   atoi 函数已在第 8 章讲述。以下是一个示范对话：

```
How many numbers to enter?
5
Enter number 1:
one
Please enter your number using digits only.  Try again.
Enter number 1:
9
Enter number 2:
3
Enter number 3:
3
Enter number 4:
33
The number must be between 1-10.  Try again.
```

```
Enter number 4:
3
Enter number 5:
7

Here is the histogram of values:
1 :
2 :
3 : ***
4 :
5 :
6 :
7 : *
8 :
9 : *
10:
```

4. 定义 CheckedArray 类，该类的对象与普通数组相似，但具有范围检查能力。假定 a 是 CheckedArray 类的对象，i 是非法索引，那么使用 a[i]会造成程序抛出异常(ArrayOutOfRangeError 类的对象)。作为本项目一部分，请自行定义 ArrayOutOfRangeError 类。注意，你的 CheckedArray 类必须恰当地重载[]操作符，具体参见附录 6。

5. 第 13 章和第 14 章介绍了栈。定义栈类来存储 char 元素。栈对象应具有固定长度；请将 size 作为创建栈对象的构造函数的一个参数。在程序中使用时，栈类的对象在以下情况下抛出异常。
   - 如应用程序试图将数据压入已经满了的栈，抛出 StackOverflowException。
   - 如应用程序试图从空栈弹出数据，抛出 StackEmptyException。

   作为本项目的一部分，请自行定义 StackOverflowException 和 StackEmptyException 类。另外，写一个恰当的测试程序。

6. (基于 Stroustrup 的著作 *The C++ Programming Language* 第 3 版提出的一个问题)写一个程序，要求函数调用函数，并总共达到 10 级调用深度。每个函数都有一个参数指定在哪一级抛出异常。main 函数要求用户输入要抛出异常的调用深度(级别)。然后，main 函数调用第一个函数。main 函数捕捉异常，显示抛出异常时的级别。不要忘记深度为 0 的情况；在这个级别上，main 必须同时抛出和捕捉异常。

   提示：可使用 10 个不同的函数，也可使用同一个函数的 10 个拷贝，但两种方法都不可取。相反，为简化代码，应使用一个 main 函数，在其中调用另一个函数，后者则递归调用自身。这时有必要限制调用深度吗？不给函数提供任何额外的参数，就能完成这道题。但如果那样无法完成，请试着为函数添加一个额外的参数。

7. 第 9 章的编程项目 7 描述了如何用一维数组的包装函数来模拟二维数组。如二维数组中的一个元素的索引无效(比如超出范围)，那道题要求打印错误消息并退出程序。请修改程序，在行或列索引无效时抛出 ArrayOutOfRangeError 异常。程序应定义 ArrayOutOfRangeError 异常类。

第 17 章　模板

所有人都是凡人，
亚里士多德是人，
所以亚里士多德是凡人。

所有 X 都是 Y，
Z 是 X，
所以 Z 是 Y。

所有猫都很淘气，
加菲猫是猫，
所以加菲猫很淘气。
——*"三段论"快速入门*

概述

本章讨论 C++模板。模板允许定义将类型名称作为参数的函数和类。这样的函数能使用不同类型的参数，而且能定义泛型类。

预备知识

17.1 节基于第 2 章～第 5 章和第 7 章的 7.1 节、7.2 节和 7.3 节的知识。不会用到有关类的任何知识。17.2 节基于第 2 章～第 7 章、第 10 章～第 12 章的知识。

17.1　用于算法抽象的模板

之前许多 C++函数定义都有一个基本算法，该算法比函数定义中给出的算法更泛化。以始见于第 5 章的 swapValues 函数为例。为方便讨论，下面复制了该函数的定义：

```
void swapValues(int& variable1, int& variable2)
{
    int temp;

    temp = variable1;
    variable1 = variable2;
    variable2 = temp;
}
```

swapValues 函数只支持 int 变量。但函数主体给出的算法同样能交换 char 类型的两个变量。要将 swapValues 函数用于 char 变量，可添加以下定义来重载函数名：

```
void swapValues(char& variable1, char& variable2)
{
    char temp;

    temp = variable1;
    variable1 = variable2;
    variable2 = temp;
}
```

但 swapValues 函数的这两个定义明显效率低，令人不满意。它们几乎一模一样。唯一区别就是一个定义三处使用了 int 类型，另一个定义在同样三处使用了 char 类型。顺着这个思路，要将 swapValues 函数应用于 double 类型的两个变量，就必须写第三个几乎完全一样的函数定义。要将 swapValues 函数应用于更多类型，必须一次又一次地重复几乎完全一样的函数定义。这要求大量打字工作，而且会使代码充斥着大量几乎完全一样的定义。我们认为，以下函数定义适合任何类型的变量：

```
void swapValues(Type_Of_The_Variables& variable1,
                Type_Of_The_Variables& variable2)
{
    Type_Of_The_Variables temp;

    temp = variable1;
    variable1 = variable2;
    variable2 = temp;
}
```

如后文所述，像这样的定义完全可行。可定义一个函数，让它应用于所有类型的变量，不过语法稍微有别于以前见过的定义。下一节将具体描述。

函数模板

图 17.1 展示了用于 swapValues 函数的 C++模板。该函数模板允许交换任何两个变量的值，两个变量可为任意类型，但前提是两个变量类型相同。定义和函数声明从下面这一行开始：

```
template<class T>
```

这通常称为**模板前缀**，它告诉编译器，后续的定义或函数声明是**模板**，T 是**类型参数**。在当前上下文中，class 一词实际的意思是 type。[①]马上就要讲到，类型参数 T 可替换成任意类型，无论该类型是不是类。在函数体中，类型参数 T 的用法和其他任何类型没有区别。

函数模板的定义实际是多个函数定义的“合集”。在如图 17.1 所示的 swapValues 函数模板中，实际为每种可能的类型名称都提供了函数定义。获取每个定义时，都是将类型参数 T 替换成实际类型名称。例如，将 T 替换成类型名称 double，就得到以下函数定义：

```
void swapValues(double& variable1, double& variable2)
{
    double temp;

    temp = variable1;
    variable1 = variable2;
    variable2 = temp;
}
```

将函数模板中的类型参数 T 替换成类型名称 int，就得到了 swapValues 的另一个定义。将 T 替换成 char，则得到另一个定义。图 17.1 的函数模板重载函数名称 swapValues，结果是每种可能的类型都存在一个稍微不同的函数定义。

---

① 事实上，ANSI 标准建议在模板前缀中使用关键字 typename，而不是 class。虽然我们也认为 typename 比 class 更合理，但 class 已在程序员心中根深蒂固。所以，本书仍然使用 class，以便与其他大多数程序员和教科书保持一致。

图 17.1 一个函数模板

```
1  // 该程序用于演示一个函数模板
2  #include <iostream>
3  using namespace std;

4  // 交换 variable1 和 variable2 的值
5  template<class T>
6  void swapValues(T& variable1, T& variable2)
7. {
8      T temp;
9
10     temp = variable1;
11     variable1 = variable2;
12     variable2 = temp;
13 }

14 int main()
15 {
16     int integer1 = 1, integer2 = 2;
17     cout << "Original integer values are "
18          << integer1 << " " << integer2 << endl;
19     swapValues(integer1, integer2);
20     cout << "Swapped integer values are "
21          << integer1 << " " << integer2 << endl;

22     char symbol1 = 'A', symbol2 = 'B';
23     cout << "Original character values are "
24          << symbol1 << " " << symbol2 << endl;
25     swapValues(symbol1, symbol2);
26     cout << "Swapped character values are "
27          << symbol1 << " " << symbol2 << endl;

28     return 0;
29 }
```

输出

```
Original integer values are 1 2
Swapped integer values are 2 1
Original character values are A B
Swapped character values are B A
```

但编译器不会真的为函数名称 swapValues 生成针对所有可能类型的定义，只是表现得像是生成了所有函数定义。针对用到的每种类型，编译器都生成单独的函数定义，但不会为没用到的任何类型生成定义。另外，无论为一种类型使用多少次模板，都只为那种类型生成一个定义。注意，图 17.1 调用了两次 swapValues 函数：一次传递 int 类型的参数，另一次传递 char 类型的参数。

分析图 17.1 以下函数调用：

```
swapValues(integer1, integer2);
```

C++编译器遇到这个函数调用时，注意到实参类型是 int，所以会用模板生成具体函数定义，并将模板中的类型参数 T 替换成类型名称 int。类似地，遇到以下函数调用时：

```
swapValues(symbol1, symbol2);
```

会注意到实参类型是 char，所以会用模板生成具体函数定义，将类型参数 T 替换成类型名称 char。

调用由函数模板定义的函数时不需要特别注意什么；调用这种函数和调用其他函数没有区别。编译器看管一切，根据函数模板自动生成函数定义。

注意，图 17.1 将函数模板定义放在程序的 main 部分之前，而且没有使用模板函数声明。函数模板可以和普通函数一样有一个声明。也许能(也许不能)将模板函数的声明和定义放在和普通函数一样的位置。但许多编译器都不支持模板函数声明，也不支持模板函数的独立编译。即使支持，不同编译器的支持细节也可能有很大区别，容易造成混乱。所以，一个保险的策略就是根本不使用模板函数声明，并确保函数模板定义和使用它的代码在同一个文件中，而且定义在使用它的代码之前。

我们说函数模板定义应放在使用该函数的文件中(也就是说，要放在含有模板函数调用的文件中)。但也可通过#include 预编译指令来包含函数模板定义。可单独在一个文件中给出函数模板定义，再在使用了模板函数的文件中通过#include 包含那个文件。这是最直观、最保险的策略。但这种方式在某些编译器上可能不支持。如发现这样行不通，请咨询懂行的人。

虽然不准备在本书代码中使用模板函数声明，但仍要稍微解释一下，并给出一些例子，供编译器支持这种功能的读者参考。

图 17.1 的函数模板用字母 T 作为类型参数。这是传统做法，但 C++语言不要求这样做。类型参数实际可为任意标识符(关键字除外)。T 是表示类型参数的很好的名称，但其他名称有时更好。例如，图 17.1 为 swapValues 提供的函数模板等价于：

```
template<class VariableType>
void swapValues(VariableType& variable1, VariableType& variable2)
{
     VariableType temp;
     temp = variable1;
     variable1 = variable2;
     variable2 = temp;
}
```

此外，函数模板也可能有多个类型参数。例如，具有两个类型参数(T1 和 T2)的一个函数模板可以像下面这样开头：

```
template<class T1, class T2>
```

但大多数函数模板都只需一个类型参数。不能有未使用的模板参数。换言之，指定的每个模板参数都必须在模板函数中用到。

函 数 模 板

函数模板的定义和声明都要用以下语句开头：

```
template<class Type_Parameter>
```

函数声明(如决定使用的话)和定义类似于其他任何普通函数，只是可用 *Type_Parameter* 代替一个类型。

例如，下面是一个函数模板的声明：

```
template<class T>
void showStuff(int sutff1, T stuff2, T stuff3);
```

该函数模板的定义可能是：

```
template<class T>
```

```
void showStuff(int sutff1, T stuff2, T stuff3)
{
    cout << stuff1 << endl
         << stuff2 << endl
         << stuff3 << endl;
}
```

本例的函数模板相当于为每个可能的类型名称都给出了函数声明和函数定义。类型名称会取代类型参数(本例是 T)。例如以下函数调用：

```
showStuff(2, 3.3, 4.4);
```

执行该函数调用时，编译器将 T 替换成类型名称 double，从而获得相应的函数定义。针对使用了该模板的每种类型，都自动生成一个独立的定义。但对于没有使用模板的类型，则不生成定义。每种类型只生成一个定义，无论使用了多少次模板。

陷阱：编译器的复杂性

视频讲解：*Issues compiling Programs with templates*

C++不允许用一般的方法分开模板定义的接口(头)文件和实现文件，模板定义必须包含在使用它的代码中。另外，如决定使用模板函数声明，声明必须在使用模板函数之前出现。

最保险的策略是根本不使用模板函数声明，同时确保函数模板定义出现在使用它的文件中，而且要在调用模板函数之前出现。不过，函数模板定义也可借助#include 预编译指令来出现。这样就可单独用一个文件存储函数模板定义，并在使用它的文件中，用#include 包含该文件。

另一个常用技术是将定义和实现全都放在头文件中。使用该技术，就只有一个头文件(.h)，没有实现文件(.cpp)。如全部代码都在头文件中，有时可使用.hpp 扩展名。最后，另一个技术是包含模板的实现文件(.cpp)而不是头文件(.h)。

有的 C++编译器对于模板的使用提出了另一些特殊要求。如编译模板遇到麻烦，请查阅编译器手册，或咨询这方面懂行的人。可能需要设置一些特殊选项，或需要重新安排模板定义和其他项目在文件中的顺序。 ■

算 法 抽 象

讨论 swapValues 函数时说过，交换两个变量的值有一种很泛化的算法。这种泛化算法适合任意类型的变量。C++用函数模板表示这种泛化算法。这是算法抽象的简单例子。**算法抽象**是指用一种更常规(泛化)的方式表示算法，忽略完全一致的细节，将重点放在算法本质上。C++设计了许多功能来支持算法抽象，函数模板是其中之一。

√ 自测题

1. 写名为 maximum 的函数模板。该函数获取同类型的两个值作为参数，返回两者中较大的(如相等，返回任何一个都可以)。给出模板的函数声明和函数定义。要在定义中使用操作符<。因此，该函数模板只适合定义了<的那些类型。请为函数声明撰写一条注释，仔细解释这一限制。

2. 目前已用过三种绝对值函数：abs、labs 和 fabs。它们唯一的区别就是参数类型。更好的方式也许是使用函数模板。请为名为 absolute 的绝对值函数给出函数模板。适合使用这个模板的类型要满足三个要求：定义了操作符<，定义了一元求反操作符(–)，而且能与常量 0 进行比较。换言之，absolute 适合任何数值类型，比如 int、long 和 double。模板的函数声明和函数定义都要给出。

3. 定义或归纳 C++的模板机制。

4. 在以下模板前缀中：

```
template<class T>
```

对参数 T 的哪一个描述是正确的？

a. T 必须是一个类。
b. T 必须不是一个类。
c. T 只能是 C++内建类型。
d. T 可为任意类型，不管是 C++内建的，还是程序员自定义的。

编程实例　泛型排序函数

第 7 章给出了简单排序算法，能对 int 类型的值构成的数组进行排序。算法用 C++函数 sort 实现，如图 7.12 所示。为方便讨论，下面复制了该 sort 函数的定义：

```
void sort(int a[], int numberUsed)
{
    int indexOfNextSmallest;
    for (int index = 0; index < numberUsed - 1; index++)
    {// 将正确的值放在 a[index]中:
        indexOfNextSmallest =
                 indexOfSmallest(a, index, numberUsed);
        swapValues(a[index], a[indexOfNextSmallest]);
        // a[0] <= a[1] <=...<= a[index]都是原始数组元素中最小的
        // 剩余的元素在剩余的位置
    }
}
```

分析上述 sort 函数定义，会发现数组基类型并不是决定性的。将函数头中的数组基类型替换成 double，就得到对 double 数组进行排序的函数。当然，另外还必须修改其中用到的辅助函数，使其支持 double 类型的数组。所以，下面分析一下 sort 函数主体内部调用的辅助函数，分别是 swapValues 和 indexOfSmallest。

上一节已介绍如何让 swapValues 函数用于任意类型，这需要把它定义成函数模板(如图 17.1 所示)。下面看看在 indexOfSmallest 函数的定义中，待排序数组的基类型是不是决定性的。为方便分析，下面复制了 indexOfSmallest 的定义：

```
int indexOfSmallest(const int a[], int startIndex, int numberUsed)
{
    int min = a[startIndex],
    int indexOfMin = startIndex;
    for (int index = startIndex + 1; index < numberUsed; index++)
        if (a[index] < min)
        {
            min = a[index];
            indexOfMin = index;
            // min 是 a[startIndex]～a[index]中最小的
        }
```

```
      return indexOfMin;
   }
```

可以看出，在 indexOfSmallest 函数的定义中，数组的基类型也是无所谓的。将粗体显示的两个 int 换成 double，indexOfSmallest 将支持基类型为 double 的数组。

要修改 sort 函数对基类型为 double 的数组进行排序，只需要将少量类型名称 int 替换成类型名称 double。另外注意，double 类型和 int 类型一样，也没有什么特殊之处。通过替换类型，还能支持其他许多类型，唯一要求就是已经为这些类型定义了操作符<。总之，这是适合使用函数模板的典型例子。将少数类型名称 int(在 sort 和 indexOfSmallest 函数中)替换成类型参数，sort 函数就能对任意类型的数组进行排序，唯一要求就是那种类型的值可用操作符<进行比较。图 17.2 给出了这样的一个函数模板。

图 17.2　泛型排序函数

```
1  // 这是 sortfunc.cpp 文件的内容
2  template<class T>
3  void swapValues(T& variable1, T& variable2)
4          <swapValues 剩余的定义与图 17.1 相同>
5  template<class BaseType>
6  int indexOfSmallest(const BaseType a[], int startIndex, int numberUsed)
7  {
8     BaseType min = a[startIndex];
9     int indexOfMin = startIndex;
10
11    for (int index = startIndex + 1; index < numberUsed; index++)
12       if (a[index] < min)
13       {
14          min = a[index];
15          indexOfMin = index;
16          //min 是[startIndex]到 a[index]之间最小的元素
17       }
18
19    return indexOfMin;
20 }
21
22 template<class BaseType>
23 void sort(BaseType a[], int numberUsed)
24 {
25 int indexOfNextSmallest;
26 for(int index = 0; index < numberUsed - 1; index++)
27    {// 将正确的值放在 a[index]中:
28       indexOfNextSmallest =
29            indexOfSmallest(a, index, numberUsed);
30       swapValues(a[index], a[indexOfNextSmallest]);
31       // a[0] <= a[1] <=...<= a[index]都是原始数组元素中最小的
32       // 剩余的元素在剩余的位置
33    }
34 }
```

注意，图 17.2 给出的 sort 函数模板可用于非数值类型的数组。图 17.3 的示范程序调用函数模板 sort 来排序字符数组。字符可用<比较。虽然操作符<在应用于字符时的确切含义在不同 C++实现中有所区别，但用<进行字母排序时，一些原则总是不变的。应用于两个大写字母时，操作符<根据字母顺序测试第一个字母是否在第二个字母之前。应用于两个小写字母时，操作符<根据字母顺序测试第一个字母是否在第二个字母之前。混合大写和小写字母，结果就没有这么确定了。但图 17.3 的程序只处理大写字母。程序调用函数模板 sort，根据字母顺序排列由大写字母构成的数组(sort 函数甚至能排序由类的对象构成的数组，但前提是要重载操作符<，使它支持那个类的对象)。

图 17.3　使用泛型排序函数

```
1  // 演示一个泛型的排序函数
2  #include <iostream>
3  using namespace std;
4
5  // sortfunc.cpp 文件定义了以下函数:
6  // template<class BaseType>
7  // void sort(BaseType a[], int numberUsed);
8  // 前条件: numberUsed <= 数组 a 的声明长度
9  // 数组元素 a[0]~a[numberUsed - 1]已经赋值
10 // 后条件: 重新排列 a[0]~a[numberUsed - 1]的值, 确保:
11 // a[0] <= a[1] <= ... <= a[numberUsed - 1]
12
13 #include "sortfunc.cpp"
14
15 int main()
16 {
17     int i;
18     int a[10] = {9, 8, 7, 6, 5, 1, 2, 3, 0, 4};
19     cout << "Unsorted integers:\n";
20     for (i = 0; i < 10; i++)
21         cout << a[i] << " ";
22     cout << endl;
23     sort(a, 10);
24     cout << "In sorted order the integers are:\n";
25     for (i = 0; i < 10; i++)
26         cout << a[i] << " ";
27     cout << endl;

28     double b[5] = {5.5, 4.4, 1.1, 3.3, 2.2};
29     cout << "Unsorted doubles:\n";
30     for (i = 0; i < 5; i++)
31         cout << b[i] << " ";
32     cout << endl;
33     sort(b, 5);
34     cout << "In sorted order the doubles are:\n";
35     for (i = 0; i < 5; i++)
36         cout << b[i] << " ";
37     cout << endl;

38     char c[7] = {'G', 'E', 'N', 'E', 'R', 'I', 'C'};
39     cout << "Unsorted characters:\n";
40     for (i = 0; i < 7; i++)
41         cout << c[i] << " ";
42     cout << endl;
43     sort(c, 7);
44     cout << "In sorted order the characters are:\n";
45     for (i = 0; i < 7; i++)
46         cout << c[i] << " ";
47     cout << endl;

48     return 0;
49 }
```

许多编译器允许实际使用该函数声明，所以可考虑取消注释状态。但函数声明并非必不可少

函数定义在文件 sortfunc.cpp 中，所以定义肯定在 main 部分之前出现

输出

```
Unsorted integers:
9 8 7 6 5 1 2 3 0 4
In sorted order the integers are:
0 1 2 3 4 5 6 7 8 9
Unsorted doubles:
5.5 4.4 1.1 3.3 2.2
In sorted order the doubles are:
1.1 2.2 3.3 4.4 5.5
Unsorted characters:
G E N E R I C
In sorted order the characters are:
C E E G I N R
```

编程提示：如何定义模板

定义如图 17.2 所示的函数模板时，首先定义能对 int 类型的数组进行排序的函数。然后创建模板，具体是将数组基类型替换成类型参数 T。这是写模板的一个很好的策略。总之，要写函数模板，先写非模板的版本，也就是写一个普通函数。全面调试这个普通函数，再将必要的类型名称替换成类型参数，从而将普通函数转换成模板。这样做有两个优点。首先，定义普通函数时，你处理的是较具体的情况，它使问题更直观。其次，由于工作分几阶段完成，所以每阶段要关心的细节会少一些。关注算法本身时，不用关心模板的语法规则。■

陷阱：为不恰当的类型使用模板

只要函数定义代码有意义，就可将模板用于任意类型。但一定要确保模板函数中的所有代码都支持那种类型，而且行为一定要正确。例如，图 17.1 的 swapValues 模板就不能用于不支持赋值操作符(或虽然支持，但行为不正确)的类型。

下面看一个更具体的例子，假定程序定义了如图 17.1 所示的模板函数 swapValues，那么不能在程序中使用以下代码：

```
int a[10], b[10];
<用于填充数组的一些代码>
swapValues(a, b);
```

上述代码不能工作，因为数组类型不支持赋值。■

√ 自测题

5. 图 7.10 展示了名为 search 的函数，它在数组中搜索指定整数。请给出 search 的函数模板版本，用它搜索任意类型的数组。模板函数声明和定义都要给出。**提示**：答案与图 7.10 的函数几乎完全相同。
6. 第 4 章的编程练习 8 要求重载 abs 函数，使同一个函数名称 abs 能支持当时学习的几种内建类型。请比较一下 abs 函数的重载方案与本章自测题 2 的模板方案。

17.2　用于数据抽象的模板

相同的财富和相同的文化机会……使我们成为同一类的成员。

——*爱德华·贝拉米 (1850—1898)，《回首 2000—1887》*

17.1 节讲到，模板使函数定义更泛化。本节要讲到，模板还能使类的定义更泛化。

类模板的语法

类模板的语法和函数模板基本一样。下面是放在模板定义之前的语句：

```
template<class T>
```

类型参数 T 在类定义中的用法与其他任何类型无异。和函数模板一样，类型参数 T 可以是任何类型；类型参数不一定要替换成类类型。和函数模板一样，可用任何(非关键字)标识符代替 T。

以下面的类模板为例。该类的对象包含一对 T 类型的值。如果 T 是 int，则对象值是一对整数；如果 T 是 char，则对象值是一对字符，依此类推。

```
// 该类表示一对 T 类型的值:
template<class T>
class Pair
{
public:
    Pair();

    Pair(T firstValue, T secondValue);

    void setElement(int position, T value);
    // 前条件: position 是 1 或 2
    // 后条件: 指定的 position 被设置成 value

    T getElement(int position) const;
    // 前条件: position 是 1 或 2
    // 返回在指定 position 的值
private:
    T first;
    T second;
};
```

定义好类模板之后就可声明该类的对象。声明必须指定要为 T 填充什么类型。例如，以下语句声明 score 对象来记录一对整数。另外还声明 seats 对象来记录一对字符：

```
Pair<int> score;
Pair<char> seats;
```

然后就可像使用其他任何对象那样使用上述两个对象。例如，以下语句将第一个球队的 score 设置成 3，将第二个球队的 score 设置成 0：

```
score.setElement(1, 3);
score.setElement(2, 0);
```

类模板的成员函数采用与普通类的成员函数一样的方式来定义。唯一的区别是，成员函数定义本身也是模板。例如，以下代码给出了成员函数 setElement 的一个恰当的定义，并给出了接收两个参数的构造函数的定义：

```
// 使用 iostream 和 cstdlib:
template<class T>
void Pair<T>::setElement(int position, T value)
{
    if (position == 1)
        first = value;
    else if (position == 2)
        second = value;
    else
    {
        cout << "Error: Illegal pair position.\n";
```

```
        exit(1);
    }
}

template<class T>
Pair<T>::Pair(T firstValue, T secondValue)
        : first(firstValue), second(secondValue)
{
    // 主体有意留空
}
```

注意，作用域解析操作符之前的类名是 Pair<T>，而非单单是 Pair。

类模板名称可用作函数参数类型。例如，以下语句声明一个函数，参数是一对整数：

```
int addUp(const Pair<int>& thePair);
// 返回 thePair 中的两个整数的和
```

注意，其中指定了要填充的类型(本例是 int)，它将取代类型参数 T。

甚至可以在函数模板中使用类模板。例如，可以不像上面那样定义具体的 addUp 函数，而是定义函数模板，使函数能支持所有数值类型：

```
template<class T>
T addUp(const Pair<T>& thePair);
// 前条件：已经为 T 类型的值定义了操作符+
// 返回 thePair 中的两个值之和
```

类模板的语法

类定义和成员函数的定义要用以下代码开头：

```
template<class Type_Parameter>
```

然后，类和成员函数的定义和其他任何普通类没有区别，只是可用 *Type_Parameter* 代替类型。

例如，下面是一个类模板定义的开头：

```
template<class T>
class Pair
{
public:
    Pair();
    Pair(T firstValue, T secondValue);
    void setElement(int position, T value);
        ...
```

之后，成员函数和重载操作符可定义成函数模板。例如，针对上述类模板，方法定义可像下面这样开头：

```
template<class T>
void Pair<T>::setElement(int position, T value)
{
    . . .
```

类 型 定 义

可为类名赋一个类型实参来特化类模板，如下例所示：

```
Pair<int>
```

然后，特化的类名(如 Pair<*int*>)可像任何类名那样使用。可用于声明对象，或用于指定形参类型。

可定义新的类型名称，让它具有与特化类模板名称(比如 Pair<*int*>)相同的含义。这种已定义的类类型名称的语法如下：

```
typedef Class_Name<Type_Argument> New_Type_Name;
```

例如：

```
typedef Pair<int> PairOfInt;
```

然后，可用类型名称 PairOfInt 声明 Pair<*int*>类型的对象，如下例所示：

```
PairOfInt pair1, pair2;
```

类型名称 PairOfInt 也可作为形参类型。

编程实例　数组类

图 17.4 展示了一个类模板的接口。该类的对象是列表。由于类定义是类模板，所以列表可包含任意类型的数据项。可创建列表对象，让它包含 int 类型的值，包含 double 类型的值，包含 string 类型的对象，或包含其他任意类型的数据项。

图 17.5 是使用了这个类模板的示范程序。虽然程序没有做太多事情，但它很好地演示了类模板的用法。理解语法细节后，就可在需要值列表的任何程序中使用这个类模板。类模板的实现在图 17.6 中给出。

注意已重载插入操作符<<，所以可输出模板类 GenericList (泛型列表) 的对象。为此，程序将操作符<<设为类的友元。要让参数具有与类相同的类型，参数类型要使用表达式 GenericList<ItemType>。例如，类型参数替换成 int，参数就具有 GenericList<int> 类型。

还要注意，重载的插入操作符<<的实现放到头文件而不是实现文件中。表面看不寻常，但在模板中使用友元函数或操作符时很常见。虽然<<似乎被定义成 GenericList 的成员，但记住友元函数实际存在于类的外部，是命名空间的一部分。在其他文件中包含类时，编译器很容易找到<<的实现。

图 17.4　类模板 GenericList 的接口

```
1  // 这是头文件 genericlist.h，它是 GenericList 类的接口
2  // GenericList 类型的对象可以是任意类型的数据项的列表，前提是已为那些类型定义了操作符<<和=
3  // 一个列表中的所有项必须具有相同类型
4  // 使用以下语句声明一个列表，最多容纳 max 个 Type_Name 类型的项：
5  // GenericList<Type_Name> theObject(max);
6
7  #ifndef GENERICLIST_H
8  #define GENERICLIST_H
9  #include <iostream>
10 using namespace std;
11
12 namespace listsavitch
13 {
14     template<class ItemType>
15     class GenericList
16     {
17     public:
```

```
18          GenericList(int max);
19          // 将对象初始化成空列表，最多能容纳 max 个 ItemType 类型的项
20
21
22          ~GenericList();
23          // 将对象占用的所有动态内存返还给自由存储(堆)
24
25          int length() const;
26          // 返回列表项数量
27
28          void add(ItemType newItem);
29          // 前条件：列表未满
30          // 后条件：newItem 添加到列表中
31
32          bool full() const;
33          // 列表满就返回 true
34
35          void erase();
36          // 从列表删除所有项，使列表为空
37
38          friend ostream& operator<<(ostream& outs,
39                                  const GenericList<ItemType>& theList)
40          {
41              for (int i = 0; i < theList.currentLength; i++)
42                      outs << theList.item[i] << endl;
43              return outs;
44          }
45          // 重载<<操作符，使其能输出列表的内容。每行输出一个项
46          // 前条件：如果 outs 是文件输出流，那么 outs 已连接到一个文件
47          //
49          // 注意重载的<<的实现放在头文件中。重载的友元模板常采用这个做法
50          // <<是友元，不是类成员，是命名空间的一部分。这个简单的
51          // 实现放在这里更合适，而不是放在 genericlist.cpp 中
52
53
54      private:
55          ItemType *item; // 指向容纳列表的动态数组的指针
56          int maxLength; // 列表允许的最大项数
57          int currentLength; // 列表目前的项数
58      };
59  } // listsavitch
60  #endif //LIST_H
```

图 17.5　使用了 GenericList 类模板的程序

```
1  // 该程序演示类模板 GenericList 的用法
2  #include <iostream>
3  #include "genericlist.h"
4  #include "genericlist.cpp"
5  using namespace std;
6  using namespace listsavitch;

7  int main()
8  {
9      genericList<int> firstList(2);
10     firstList.add(1);
11     firstList.add(2);
12     cout << "firstList = \n"
13          << firstList;

14     genericList<char> secondList(10);
15     secondList.add('A');
16     secondList.add('B');
17     secondList.add('C');
18     cout << "secondList = \n"
19          << secondList;
```

由于已包含 genericlist.cpp，所以只需编译这个文件(有 main 部分的这个)

```
20      return 0;
21  }
```

输出

```
firstList =
1
2
secondList =
A
B
C
```

图 17.6　GenericList 的实现

```
1  // 这是实现文件 genericlist.cpp，它包含类模板 GenericList 的实现
2  // 类模板 GenericList 的接口在头文件 genericlist.h 中给出
3
4
5  #ifndef GENERICLIST_CPP
6  #define GENERICLIST_CPP
7  #include <iostream>
8  #include <cstdlib>
9  #include "genericlist.h" // 这行语句本不需要，但 genericlist.h 中的#ifndef 确保了它的安全性
10
11 using namespace std;
12
13 namespace listsavitch
14 {
15     // 使用 cstdlib:
16     template<class ItemType>
17     GenericList<ItemType>::GenericList(int max) : maxLength(max), currentLength(0)
18
19     {
20         item = new ItemType[max];
21     }
22
23     template<class ItemType>
24     GenericList<ItemType>::~GenericList()
25     {
26         delete [] item;
27     }
28
29     template<class ItemType>
30     int GenericList<ItemType>::length() const
31     {
32        return (currentLength);
33     }
34
35     // 使用 iostream 和 cstdlib:
36     template<class ItemType>
37     void GenericList<ItemType>::add(ItemType newItem)
38     {
39         if ( full() )
40         {
41             cout << "Error: adding to a full list.\n";
42             exit(1);
43         }
44         else
45         {
46             item[currentLength] = newItem;
47             currentLength = currentLength + 1;
48         }
49     }
50
51     template<class ItemType>
52     bool GenericList<ItemType>::full() const
53     {
54         return (currentLength == maxLength);
```

```
55      }
56
57      template<class ItemType>
58      void GenericList<ItemType>::erase()
59      {
60          currentLength = 0;
61      }
62  }// listsavitch
63  #endif // GENERICLIST_CPP 注意所有模板定义都封装在#ifndef ... #endif 内
```

编译图 17.4、图 17.5 和图 17.6 的代码时要注意，安全编译方案是像本例展示的那样，在实际使用模板类和模板函数的定义之前用`#include` 包含这些定义。这样实际只需编译图 17.5 的那个文件。一定要用`#ifndef #define #endif` 机制防止多次包含同一文件

需多做一点事情才能将插入操作符<<的实现从头文件中拿走。<<操作符必须前置声明(forward declaration)，在它之前又要求 `GenericList` 类的前置声明。图 17.7 展示了 genericlist.h 需进行的改动，图 17.8 则展示了对 genericlist.cpp 的改动，后者只是新增了额外的实现。

图 17.7 GenericList 类模板的接口(不含实现)

```
1  // 这个版本将重载<<操作符的实现移至.cpp 文件，
2. // 但要求添加一些前置声明。
3  #ifndef GENERICLIST_H
4  #define GENERICLIST_H
5  #include <iostream>
6  using namespace std;
7
8  namespace listsavitch
9  {
10     template<class ItemType>
11     class GenericList;
12     // 有了 GenericList 模板类的这个前置声明，之后的
13     // 友元声明才能生效
14
15     template<class ItemType>
16     ostream& operator <<(ostream& outs, const GenericList<ItemType>& theList);
17     // 针对下面的 GenericList 类中的友元<<定义，
18     // 这个前置声明是必须的。只能在这里定义，
19     // 因为<<不是类的成员。
20
21     template<class ItemType>
22     class GenericList
23     {
24         类剩余部分和图 17.4 一样，只是下面的重载操作符
25         没有实现代码，而且添加了一个额外的<>。
26
27         friend ostream& operator << <>(ostream& outs,
28                                        const GenericList<ItemType>& theList);
29         // 重载<<操作符以便输出
30         // 列表的内容。
31         // 注意操作符(或函数)名之后的<>是必须的！
32         // 实现代码放在 genericlist.cpp(图 17.8)中。
33     };
34 }//listsavitch
35 #endif //GENERICLIST_H
```

图 17.8　GenericList 的实现(含重载操作符的实现)

```
1  // 这是实现文件： genericlist.cpp，
2  // 用于实现类模板 GenericList，
3  // 类模板 GenericList 的接口在
4  // 头文件 genericlist.h 中。
5  #ifndef GENERICLIST_CPP
6  #define GENERICLIST_CPP
7  #include <iostream>
8  #include <cstdlib>
9  #include "genericlist.h" // 这行语句本不需要，但 genericlist.h 中
10                          // 的#ifndef 确保了它的安全性
11 using namespace std;
12
13 namespace listsavitch
14 {
15       该文件剩余部分和图 17.6 一样，只是
16       新增了<<的实现。
17    template<class ItemType>
18    ostream& operator <<(ostream& outs, const GenericList<ItemType>& theList)
19    {
20        for (int i = 0; i < theList.currentLength; i++)
21            outs << theList.item[i] << endl;
22        return outs;
23    }
24 }//listsavitch
25 #endif   // GENERICLIST_CPP 注意所有模板定义都封装在#ifndef ... #endif 内
```

√ 自测题

7. 为“类模板的语法”一节讨论的类模板 Pair 给出成员函数 getElement 的定义。

8. 为“类模板的语法”一节讨论的类模板 Pair 给出无参构造函数的定义。

9. 给出名为 HeterogeneousPair 的模板类的定义，它类似于“类模板的语法”一节讨论类模板 Pair，只是在使用 HeterogeneousPair 时，第一个位置和第二个位置可存储不同类型的值。请使用两个类型参数 T1 和 T2；第一个位置的所有项都是 T1 类型，第二个位置的所有项都是 T2 类型。在模板类 HeterogeneousPair 中，模板类 Pair 的单一赋值函数 setElement 应被替换为两个赋值函数，分别命名为 setFirst 和 setSecond。类似地，在模板类 HeterogeneousPair 中，模板类 Pair 的单一取值函数 getElement 应被替换成两个取值函数，分别命名为 getFirst 和 getSecond。

10. 以下说法是否正确？

    对于模板和非模板类，友元用法完全一样。

小　　结

- 使用函数模板，可定义获取类型参数的函数。
- 使用类模板，可贯穿整个类使用类型参数。

自测题答案

1. 函数声明：

```
template<class T>
T maximum(T first, T second);
// 前条件：已经为类型 T 定义了操作符<
// 返回 first 和 second 中较大的
```

定义：

```
template<class T>
T maximum(T first, T second)
{
    if (first < second)
        return second;
    else
        return first;
}
```

2. 函数声明：

```
template<class T>
T absolute(T value);
// 前条件：定义了表达式 x < 0 和-x，其中的 x 属于 T 类型
// 返回它的参数值的绝对值
```

定义：

```
template<class T>
T absolute(T value)
{
    if (value <0)
        return -value;
    else
        return value;
}
```

3. 模板允许函数和类使用类型名称作为参数。

4. d. 任意类型，无论是基元类型(C++提供的 int，float，double 等)，还是用户自定义类型(class 或 struct 类型，enum 类型，已定义的数组类型)。

5. 函数声明和函数定义如下所示。它们与图 7.10 的版本几乎完全一样，只是在参数列表中，int 的两个实例被更改为 BaseType。

函数声明：

```
template<class BaseType>
int search(const BaseType a[], int numberUsed, BaseType target);
// 前条件：numberUsed <= a 的声明长度
// 另外，a[0]～a[numberUsed -1]已被赋值
// 返回 a[index] == target 的第一个索引，前提是有这样的一个索引，否则返回-1
```

定义：

```
template<class BaseType>
int search(const BaseType a[], int numberUsed, BaseType target)
{
    int index = 0, found = false;
    while ((!found) && (index < numberUsed))
        if (target == a[index])
            found = true;
        else
            index++;

    if (found)
```

```
        return index;
    else
        return -1;
}
```

6. 函数重载只适合重载时规定的类型。某些类型也许能自动转换成重载所支持的类型，但行为可能并不是你所预期的。相反，模板方案适合调用时指定的任何类型，唯一要求就是已为那种类型定义了操作符<。

7.
```
// 使用 iostream 和 cstdlib:
template<class T>
T Pair<T>::getElement(int position) const
{
    if (position == 1)
        return first;
    else if (position == 2)
        return second;
    else
    {
        cout << "Error: Illegal pair position.\n";
        exit(1);
    }
}
```

8. 默认的初始值不好确定，所以这个构造函数什么事情都不做。但是，它确实允许你在不给出任何构造函数参数的前提下声明(未初始化的)对象。

```
template<class T>
Pair<T>::Pair()
{
    // 什么都不做
}
```

9.
```
// 用于表示一对值的类，第一个值的类型是 T1，第二个的类型是 T2:
template<class T1, class T2>
class HeterogeneousPair
{
public:
    HeterogeneousPair();
    HeterogeneousPair(T1 firstValue, T2 secondValue);
    void setFirst(T1 value);
    void setSecond(T2 value);
    T1 getFirst() const;
    T2 getSecond() const;
private:
    T1 first;
    T2 second;
};
```

成员函数定义如下所示：

```
template<class T1, class T2>
HeterogeneousPair<T1, T2>::HeterogeneousPair()
{
    // 什么都不做
}

template<class T1, class T2>
HeterogeneousPair<T1, T2>::HeterogeneousPair
    (T1 firstValue, T2 secondValue)
        : first(firstValue), second(secondValue)
{
    // 主体有意留空
}

template<class T1, class T2>
T1 HeterogeneousPair<T1, T2>::getFirst() const
{
    return first;
}

template<class T1, class T2>
T2 HeterogeneousPair<T1, T2>::getSecond() const
```

```
{
    return second;
}

template<class T1, class T2>
void HeterogeneousPair<T1, T2>::setFirst(T1 value)
{
    first = value;
}

template<class T1, class T2>
void HeterogeneousPair<T1, T2>::setSecond(T2 value)
{
    second = value;
}
```

10. 正确。

编 程 练 习

编程练习一般只需写很小的程序，运用本章提到的编程概念。

1. 为一个函数写函数模板。函数参数包括一个部分填充的数组和数组基类型的一个值。如指定的值在部分填充的数组中，函数返回包含那个值的第一个索引变量的索引；如指定的值不在数组中，函数返回-1。数组基类型是一个类型参数。注意需要两个参数来给出部分填充的数组：一个指定数组，另一个指定使用的索引变量数量。另外，写合适的测试程序来测试该函数模板。

2. 为图 14.8 的迭代二叉搜索写模板版本。规定并讨论对模板参数类型的要求。

3. 为图 14.6 的递归二叉搜索写模板版本。规定并讨论对模板参数类型的要求。

编 程 项 目

编程项目要求综合运用多方面的知识来解决问题，程序一般比编程练习大，解题方式多样化。

1. 重写图 17.4 和图 17.6 给出的类模板 GenericList 的定义，使其更泛化。该版本支持一项新功能，允许按顺序遍历列表项。肯定有一个项是当前项。可请求当前项，将当前项更改为下一项，将当前项更改为上一个项，将第一项设为当前项以便从列表起始处开始，以及请求列表第 *n* 项。需添加以下成员：一个附加的成员变量，用于记录当前项在列表中的位置；一个成员函数，将当前项作为一个值来返回；一个成员函数，使下一项成为当前项；一个成员函数，使上一项成为当前项；一个成员函数，使列表第一项成为当前项；以及一个成员函数，以 *n* 为参数，返回列表第 *n* 项(项编号方案与数组相同，所以第一项编号为 0，下一项编号为 1，以此类推)。
注意在某些情况下，上述某些函数操作不可行。例如，空列表就没有第一项。另外，在任何列表中，最后一项之后都没有“下一项”。一定要测试空列表，确保能正确处理这种列表。还必须测试列表的开头与结尾，正确处理这些情况。写合适的测试程序来测试该类模板。

2. 写一个函数模板，函数的参数包括一个数据项列表以及可能在列表中的一个项。如指定项在列表中，函数就返回该项首次出现的位置；如不在，函数返回-1。列表第一个位置是位置 0，下一个位置是位置 1，依此类推。列表项的类型用一个类型参数指定。请使用你在编程项目 1 定义的类模板 GenericList。写合适程序来测试该函数模板。

3. 重做第 7 章的编程项目 3，这次使 deleteRepeats 成为模板函数，它通过一个类型参数来指定数组基类型。先完成非模板版本会有帮助。也就是说，最好先完成第 7 章的编程项目 3(如果还没有完成的话)。

4. 图 17.3 的模板函数使用选择排序算法对数组进行排序。写类似模板函数来排序数组，但这次使用第 7 章的编程项目 6 描述的插入排序算法。还没有完成那个项目就先完成(先做一个非模板的版本)。

5. (本项目要求你知道什么是栈，并知道如何使用动态数组。栈在第 14 章讨论，动态数组在第 9 章讨论。所以，要做这个项目，必须掌握第 9 章和第 14 章的内容)。

写栈类的模板版本。为栈中存储的数据的类型使用类型参数。使用动态数组，允许栈自由扩展以容纳任意数量的数据项。

6. 为实现优先级队列的类实现模板版本。队列在第 13 章讨论，优先级队列将在第 18 章讨论。简单地说，优先级队列是总按优先级排列的数据项列表。添加到列表的每一项都要求关联的优先级值。本题假定优先级是整数，0 代表最高优先级；值越大，优先级越低。从队列中删除一项时，删除的是具有最高优先级的那一项。

   优先级队列的 add 函数应获取一个泛型类型和一个整数优先级作为参数。在下例中，泛型类型是 char，我们在队列中添加了三个数据项：

```
q.add('X', 10);
q.add('Y', 1);
q.add('Z', 3);
```

   remove 函数应从优先级队列中删除具有最高优先级的项，并将其返回。以上例为基础，执行 remove() 将得到以下结果：

```
cout << q.remove();      // 输出 Y(优先级 1)
cout << q.remove();      // 输出 Z(优先级 3)
cout << q.remove();      // 输出 X(优先级 10)
```

   在队列中添加基于不同顺序(比如升序、降序、混合序)的优先级数据，从而测试你的队列。可用自己选择的一个列表(比如向量、数组、链表或者本章描述的 GenericList)来存储数据项，从而实现自己的优先级队列。获得队列之后，可用 remove 函数来执行线性搜索，获得具有最小整数值的数据项。在未来的课程中，你可能会学到名为“堆”(heap)的一种数据结构，它提供了一种更有效的方式来实现优先级队列。

7. 视频讲解：*Solution to Programming Project 17.7*

   写一个基于模板的类，它用于实现一个数据项集合。在集合(set)中，每个数据项都独一无二，都只出现一次。在内部，可用你选择的数据结构(比如列表、向量、数组等)表示集合。但类对外应支持以下函数。

   a. 在集合中添加一个新项。如项已在集合中，什么事情都不会发生。
   b. 从集合删除一个项。
   c. 返回集合中的项的数量。
   d. 判断一个项是否集合的成员。
   e. 返回指向一个动态创建的数组的指针，数组包含集合的每一项。函数调用者自己负责回收内存。

   为测试你的类，请创建不同数据类型(比如字符串、整数或其他类)的多个集合。如在集合中添加的是对象，可能需要在对象所属的类中重载==和!=操作符，使你的基于模板的集合类能正确判断成员资格。

8. 这个项目要求先完成刚才的编程项目 7 和第 14 章的编程项目 8。第 14 章的编程项目 8 要求写程序来找出集合项的所有排列组合。修改那个程序，先创建编程项目 7 所定义的模板集合类的实例，再就这个实例找出集合项的所有排列组合。可考虑利用这个基于模板的集合类来简化排列组合算法本身。

   算法要求存储一个列表集合。C++允许使用基于模板的集合类来创建集合列表。例如，在 mySet<vector<T> >所定义的集合中，包含类型 T 的一个向量。最后两个>之间的空格是必须的，否则编译器可能会被混淆。无空格的 mySet<vector<T>>可能造成编译错误(除非是 C++11 或更高版本)。

   程序应针对几个不同大小的集合，打印它们的数据项的所有排列组合。另外，应试验在集合中包含不同类型的数据(例如，可以创建 3 个整数的集合、4 个字符串的集合以及 5 个 double 值的集合，再分别就这些集合计算排列组合)。

9. 本章只为模板类使用了一个类型参数。C++允许指定多个类型参数。例如，以下代码指定类接收两个类型参数：

```
template<class T, class V>
class Example
{
    ...
}
```

创建该类的实例必须指定两个数据类型，例如：

```
Example<int, char> demo;
```

创建 Map 类将键(key)映射到值(value)。 键值数据类型用类型参数分别指定。映射是简单数据库的基础。例如，要从员工 ID 映射到员工姓名，键可为整数，值可为字符串。类要提供函数做以下事情：

1. 向映射添加新的键/值对。
2. 给定键，将值设为新值。
3. 给定键，删除键/值对。
4. 给定键，检查是否存在键/值对。
5. 给定键，获取值。

使用你希望的任何数据类型实现映射。写 main 函数测试类，用示例数据执行所有函数。

第 18 章　标准模板库

图书馆的馆藏[①]并非一日而就，而是逐渐发展起来的。

——*奥古斯 J・比勒尔*[②] *(354—430)*

概述

由于有大量标准数据结构用于容纳数据，人们觉得有必要为这些数据结构提供标准的、可移植的实现。**标准模板库**(Standard Template Library，STL)包含了栈、队列和其他许多标准数据结构的实现。在 STL 的背景下讨论时，这些数据结构通常称为**容器类**，因其用于容纳数据集合。第 8 章已接触到了 STL。我们描述了 vector 模板类，它是 STL 容器类之一。本章要介绍 STL 包含的一些基本类。因篇幅所限，不可能涉及 STL 的方方面面。不过，本章内容足以让你开始使用一些基本 STL 容器类。

STL 是惠普公司的两个人 Alexander Stepanov 和 Meng Lee 在 Stepanov Lee 和 David Musser 三人的研究基础上开发的。它是用 C++语言写的库集合。虽然 STL 不是核心 C++语言的一部分，但它是 C++标准的一部分。因此，宣称符合 C++标准的任何 C++实现都必须包含 STL。事实上，完全可将 STL 视为 C++语言的一部分。

正如名称所暗示的，STL 中的类都是模板类。典型 STL 容器类会通过一个类型参数来指定类中存储的数据类型。STL 容器类普遍使用了迭代器。迭代器是一种特殊对象，它简化了遍历容器中的所有数据的过程。13.1 节已讲述了迭代器的概念，当时讨论了如何将指针作为迭代器使用。在继续本章的学习之前，先弄懂那一节的内容是很有必要的。如果你忘记了 vector 模板类，还应阅读第 8 章的 8.3 节，其中讨论了 STL 的 vector 模板类。

STL 还包含许多重要的泛型算法的实现，比如搜索和排序算法。这些算法都作为模板函数实现。讨论了容器类之后，将对部分算法实现进行说明。

和其他 C++库(比如<iostream>)不同，STL 中的类和算法都是**泛型**，这是模板类或模板函数的另一种说法。

预备知识

本章基于第 2 章～第 13 章、第 15 章和第 17 章的内容。

18.1　迭　代　器

白兔戴上了眼镜，问：“我应该从哪儿开始呢？陛下。”

“从开始的地方开始吧，一直读到末尾，然后停。”国王郑重其事地说。

——*刘易斯・卡洛尔，《爱丽丝漫游仙境》*

---

① library 有“图书馆”和“库”的意思。——译注

② 古罗马帝国时期天主教思想家，欧洲中世纪基督教神学、教父哲学的重要代表人物。在罗马天主教系统，他被封为圣人和圣师。对于新教教会，特别是加尔文教派，他的理论是宗教改革的救赎和恩典思想的源头。——译注

第 8 章讨论了向量，它是 STL 的容器模板类之一。迭代器是指针的泛化形式(第 13 章简单讲述了如何将指针作为迭代器使用)。本节展示如何将迭代器用于向量。18.2 节描述的其他容器模板类采取相同方式使用迭代器。所以，本节学到的迭代器知识适合大范围的容器，而非只适合向量。这反映了 STL 体系的一项基本原则：迭代器的语义、命名及语法在所有容器类型中统一。首先复习和讨论 using 声明，以后讨论迭代器和 STL 时将频繁用到它。

using 声明

继续本节和本章的学习之前，有必要复习一下第 12 章的 12.2 节。

假定 my_function 是命名空间 my_space 中定义的函数。以下 using 声明允许直接使用标识符 my_function 表示命名空间 my_space 中定义的 my_function 版本：

```
using my_space::my_function;
```

在这个 using 声明的作用域内，像 my_function(1, 2)这样的表达式实际就是 my_space::my_function(1, 2)。换言之，在 using 声明的作用域内，标识符 my_function 总是表示 my_space 中定义的那个版本的 my_function，而不是其他任何命名空间中定义的 my_function。

讨论迭代器时，经常要向另一级别应用::操作符。以下表达式经常出现：

```
using std::vector<int>::iterator;
```

在本例中，标识符 iterator 命名了一种类型。所以，在这个 using 预编译指令的作用域中，以下语句是允许的：

```
iterator p;
```

上述语句声明具有 iterator 类型的 p。何为 iterator 类型？它是在 vector<int>类的定义中定义的。何为 vector<int>类？它是在 std 中定义的。(后文会全面解释 iterator 类型，目前只关心对 using 预编译指令的解释)。

你可能表示异议，做了这么多事情，感觉是在做无用功。除了命名空间 std，其他命名空间都没有定义 vector<int>。也许吧。但现在或将来，另一些命名空间可能定义名为 vector<int>的类。你可能继续表示异议，因为根本没听说过还能在一个类中定义一个类型。虽然本书不会讨论这样的定义，但它们完全可能，在 STL 中还很常见。必须知道如何使用这些类型，即使你不去定义这些类型。

总之，对于以下 using 预编译指令：

```
using std::vector<int>::iterator;
```

在该 using 预编译指令的作用域内，标识符 iterator 是在 vector<int>类中定义的名为 iterator 的类型。vector<int>类在 std 命名空间中定义。

迭代器基础

迭代器(iterator)是指针的泛化形式。事实上，它经常就是用指针来实现的。但之所以要设计迭代器，其宗旨正是为了隐藏实现细节，提供在所有容器类中都一致的迭代器接口。

每种数据类型都有自己的指针类型；类似地，每个容器类都有自己的迭代器类型。就像指针类型对于相应数据类型的所有动态变量都具有相同行为，每种迭代器类型也是如此。不过，每种迭代器只能用于它自己的容器类。

虽然迭代器不是指针，但不妨把它当作指针使用。和指针变量相似，迭代器变量定位(指向)容器中的数据项。可用以下应用于迭代器对象的重载操作符来操纵迭代器。

- 前递增和后递增操作符++，将迭代器移至下一个数据项。
- 前递减和后递减操作符--，将迭代器移至上一个数据项。
- 相等操作符==和不相等操作符!=，测试两个迭代器是否指向同一个数据位置。
- 提领操作符*，假如 p 是迭代器变量，使用*p 就能访问位于 p 处的数据(由 p 指向的数据)。这种访问方式可以是只读、只写或可读可写的——具体由特定容器类决定。

注意，所有迭代器都拥有上述全部操作符。但 vector 模板类是特殊容器，它的迭代器除了拥有所有这些操作符，还有其他操作符。

容器类的一些成员函数用于启动迭代器过程。毕竟，新迭代器变量并不定位(指向)容器中的任何数据。许多容器类(包括 vector 模板类)都提供了以下成员函数来返回迭代器对象(迭代器值)，指向数据结构中的特定数据元素。

- c.begin()返回容器 c 的迭代器，该迭代器指向容器 c 的“第一个”数据项。
- c.end()返回某个东西来测试迭代器在什么时候越过容器 c 的最后一个数据项。第 13 章讲过 NULL，它测试指针在什么时候越过链表最后一个节点。迭代器 c.end()等价于 NULL。所以，迭代器 c.end()不定位任何数据项，只是一种尾部标志或“哨兵值”。

对于许多容器类，都可利用上述工具写一个标准的 for 循环来遍历容器对象 c 中的所有元素。如下所示：

```
// p 是容器对象 c 的那个类型的迭代器变量
for (p = c.begin(); p != c.end(); p++)
    处理*p // *p 是当前数据项
```

掌握这些基础知识后，接下来研究 vector 模板容器类的具体设置。

图 18.1 展示了如何为 vector 模板类使用迭代器。注意 STL 的每种容器类型都有自己的迭代器类型，不过所有迭代器的基本用法是一样的。由 int 构成的一个 vector 具有以下类型的迭代器：

```
std::vector<int>::iterator
```

另一个容器类是 list 模板类。由 int 构成的 list 具有以下类型的迭代器：

```
std::list<int>::iterator
```

图 18.1 用于 Vector 的迭代器

```
1  // 该程序用于演示 STL 迭代器
2  #include <iostream>
3  #include <vector>
4  using std::cout;
5  using std::endl;
6  using std::vector;
7
```

```
 8  int main()
 9  {
10     vector<int> container;
11     for (int i = 1; i <= 4; i++)
12        container.push_back(i);
13     cout << "Here is what is in the container:\n";
14     vector<int>::iterator p;
15     for (p = container.begin(); p != container.end(); p++)
16        cout << *p << " ";
17     cout << endl;
18     cout << "Setting entries to 0:\n";
19     for (p = container.begin(); p != container.end(); p++)
20         *p = 0;
21
22     cout << "Container now contains:\n";
23     for (p = container.begin(); p != container.end(); p++)
24        cout << *p << " ";
25     cout << endl;
26     return 0;
27  }
```

示范对话

```
Here is what is in the container:
1 2 3 4
Setting entries to 0:
Container now contains:
0 0 0 0
```

图 18.1 的程序特化了类型名称 iterator，将它应用于 int 向量的迭代器。程序使用的类型名称 iterator 是在模板类 vector 中定义的。所以，为了将模板类 vector 特化为 int，并使用 vector<int>的迭代器类型，要像下面这样定义：

```
std::vector<int>::iterator;
```

由于 vector 定义将 vector 这个名称放到 std 命名空间中，所以完整的 using 声明如下所示：

```
using std::vector<int>::iterator;
```

图 18.1 的以下几行代码展示了用于 vector(或其他任何容器类)的迭代器的基本用法：

```
vector<int>::iterator p;
for (p = container.begin(); p != container.end(); p++)
    cout << *p << " ";
```

记住，container 具有 vector<int>类型。

向量 v 可视为其数据元素的线型排列。第一个数据元素是 v[0]，第二个是 v[1]，依此类推。迭代器 iterator p 是能**定位**上述任何元素的对象(可认为 p 指向某个元素)。迭代器可将它的位置从一个元素移至另一个。假定 p 当前定位 v[7]，p++会使 p 重定位到 v[8]。这就允许迭代器遍历向量，从第一个元素遍历到最后一个。但这需要能够找到第一个元素，而且需要知道在什么时候遇到最后一个元素。

可用操作符==判断两个迭代器是否定位到(指向)同一个位置。所以，假如已知一个迭代器指向第一个、最后一个或其他某个元素，就可以测试另一个迭代器是否指向第一个、最后一个或其他某个元素。

假如 p1 和 p2 是两个迭代器，以下表达式就会(而且只会)在 p1 和 p2 定位到同一个元素时为 true(这和指针相似；假如 p1 和 p2 是指针，两个指针指向同一个数据项时表达式为 true)：

```
p1 == p2
```

!=和==相反，以下表达式在 p1 和 p2 不定位到同一个元素时为 true：

```
p1 != p2
```

成员函数 begin()将迭代器定位到容器的第一个元素。对于向量和其他许多容器类，成员函数 begin()返回定位到第一个元素的迭代器(对于向量 v，第一个元素是 v[0])。所以，以下语句将迭代器变量 p 初始化成定位到第一个元素的迭代器：

```
vector<int>::iterator p = v.begin();
```

所以，遍历向量 v 的所有元素的基本 for 循环如下：

```
vector<int>::iterator p;
for (p = v.begin(); 布尔表达式; p++)
  在位置 p 采取的操作;
```

作为停止条件的布尔表达式如下：

```
p == v.end()
```

成员函数 end()返回哨兵值。检查哨兵值就知道迭代器是否越过最后一个元素。假如 p 已定位到最后一个元素，在执行 p++之后，测试条件 p==v.end()就从 false 变成 true。因此，添加了正确“*布尔表达式*”的 for 循环变成以下形式：

```
vector<int>::iterator p;
for (p = v.begin(); p != v.end(); p++)
  在位置 p 采取的操作;
```

注意，除非 p 的位置跳过最后一个元素，否则 p != v.end()不会从 true 变成 false。那时，v.end()将不定位任何元素。v.end()是特殊值，作为哨兵值使用。虽然 v.end()不是普通迭代器，但完全可以使用==和!=将它和普通迭代器进行比较。v.end()值类似于标记链表结束的 NULL 值(参见第 13 章)。

迭　代　器

迭代器是和容器配合使用的对象，允许访问容器中的元素。迭代器是指针的泛化形式，而且==，!=，++和--等操作符在应用于迭代器时，其行为和应用于指针时相同。利用迭代器遍历容器中所有元素的基本方式如下：

```
STL_Container<type>::iterator p;
for (p = container.begin(); p != container.end(); p++)
   处理位置 p 处的元素;
```

其中，*STL_Container* 是容器类的名称(比如 vector)，*type* 是存储的数据项的类型。成员函数 begin()返回定位到第一个元素的迭代器。成员函数 end()返回哨兵值，标志容器最后一个元素之后的位置。

图 18.1 以下 for 循环将上述技术应用于向量 container：

```
vector<int>::iterator p;
for (p = container.begin(); p != container.end(); p++)
  cout << *p << " ";
```

在迭代器 p 的位置采取的操作如下：

```
cout << *p << " ";
```

提领操作符*为 STL 容器迭代器进行了重载，所以*p 能生成位置 p 处的元素。具体地说，对于向量容器，*p 将生成位于迭代器 p 处的元素。因此，上述 cout 语句输出迭代器 p 处的元素，整个 for 循环输出 container 向量容器中的全部元素。

提领操作符*p 总是生成迭代器 p 处的元素。某些情况下，*p 生成的是只读访问，不允许更改。另一些情况下，元素既可访问，也可更改。对于向量，*p 允许更改位于 p 处的元素。图 18.1 以下 for 循环对此进行了演示：

```
for (p = container.begin(); p != container.end(); p++)
    *p = 0;
```

这个 for 循环遍历向量 container 中的所有元素，将所有元素更改为 0。

提　　领

将提领操作符*p 应用于迭代器 p，将生成位于迭代器 p 的元素。对于某些 STL 容器类，*p 生成只读访问，不允许更改。另一些 STL 容器类则允许读取和更改元素。

编程提示：使用 auto 简化变量声明

使用模板和迭代器时，auto 关键字能使代码更易读。声明迭代器可能相当冗长：

```
vector<int>::iterator p = v.begin();
```

用 auto 就能大幅简化：

```
auto p = v.begin();
```

■

陷阱：编译器问题

有的编译器处理迭代器声明时仍然有问题。有多种不同的方式声明迭代器。例如，之前使用的是以下语句：

```
using std::vector;
 ...
vector<char>::iterator p;
```

除此之外，如果代码只用一个类型的迭代器，可以使用以下语句：

```
using std::vector<char>::iterator
...
iterator p;
```

还可使用以下语句，虽然它并不是特别好——因为会将 std 命名空间中的所有名称都引入

当前范围，使发生名称冲突的概率增大：

```
using namespace std;
  ...
vector<char>::iterator p;
```

除此之外，还有其他类似的变化形式。理想的编译器应接受所有形式。但我们发现有的编译器只接受特定形式。如一种声明形式不适合你的编译器，请尝试另一种。 ■

√ 自测题

1. 假定 v 是向量，v.begin()返回什么？v.end()返回什么？
2. 假定 p 是向量对象 v 的迭代器，*p 是什么？
3. 假定 v 是 int 向量。请写 for 循环输出 v 除了第一个元素之外的所有元素。

迭代器的种类

不同容器具有不同种类的迭代器。迭代器按操作分类。向量迭代器最常规；换言之，所有操作都适合向量迭代器。所以，将再次用向量容器演示迭代器。本例用向量演示用于**递减**和**随机访问**的迭代器操作符。图 18.2 的程序使用了向量对象 container 和迭代器 p。

图 18.2 双向迭代器和随机访问迭代器的用法

```
1  // 本程序演示了双向迭代器和随机访问迭代器
2  #include  <iostream>
3  #include <vector>
4  using std::cout;
5  using std::endl;
6  using std::vector;
7
8  int main()
9  {
10     vector<char> container;
11     container.push_back('A');
12     container.push_back('B');
13     container.push_back('C');
14     container.push_back('D');
15     for (int i = 0; i < 4; i++)
16         cout << "container[" << i << "] == "
17              << container[i] << endl;
18     vector<char>::iterator p = container.begin();
19     cout << "The third entry is " << container[2] << endl;
20     cout << "The third entry is " << p[2] << endl;
21     cout << "The third entry is " << *(p + 2) << endl;
22     cout << "Back to container[0].\n";
23     p = container.begin();
24     cout << "which has value " << *p << endl;
25     cout << "Two steps forward and one step back:\n";
26     p++;
27     cout << *p << endl;
28     p++;
29     cout << *p << endl;
30     p--;
```

迭代器的三种不同表示法

这种表示法是向量和数组特有的（第 19 行 container[2]）

这两种表示法适合任何随机访问迭代器（第 20、21 行 p[2] 和 *(p + 2)）

这是递减操作符。适合任何双向迭代器（第 30 行 p--）

```
31     cout << *p << endl;
32     return 0;
33  }
```

示范对话

```
container[0] == A
container[1] == B
container[2] == C
container[3] == D
The third entry is C
The third entry is C
The third entry is C
Back to container[0].
which has value A
Two steps forward and one step back:
B
C
B
```

图 18.2 使用了**递减操作符**，那一行加了底纹。和你想的一样，p--将迭代器 p 移至前一个位置；递减操作符--与递增操作符++相似，但后者朝相反方向移动。

递增操作符和递减操作符可采用前缀和后缀表示法，例如++p 或 p++。除了更改 p，它们还返回一个值。返回值的细节和操作符应用于 int 变量时完全一样。采用前缀表示法会先更改变量，再返回更改过的值；采用后缀表示法，则先返回值，再更改变量。一般不将递增操作符和递减操作符作为返回值的表达式使用，而只是用它们更改变量值。

图 18.2 以下代码行演示了如何使用向量迭代器对向量(比如 container)的元素进行随机访问：

```
iterator p = container.begin();
cout << "The third entry is " << container[2] << endl;
cout << "The third entry is " << p[2] << endl;
cout << "The third entry is " << *(p + 2) << endl;
```

随机访问意味着可直接跳到任何特定元素。已通过 container[2]对向量进行了随机访问，它就是数组和向量的标准方括号操作符。现在可为迭代器使用一样的方括号表示法。例如，表达式 p[2]访问索引为 2 的元素。

表达式 p[2]和*(p + 2)完全等价。与第 9 章讲的指针算法相似，(p + 2)表示在 p 的位置之后的第二个位置。由于 p 在上述代码中指向第一个位置(索引 0)，所以(p + 2)指向第三个位置(索引 2)。表达式(p + 2)返回一个迭代器。表达式*(p + 2)则提领那个迭代器。当然，可将 2 替换成不同的非负整数，获取指向不同元素的指针。

一定要注意，无论 p[2]还是(p + 2)，都不会更改迭代器变量 p 中的迭代器的值。表达式(p + 2)返回指向另一个位置的迭代器，p 保持不变。p[2]也如此。还要注意，p[2]和(p + 2)的含义取决于 p 中的迭代器的位置。例如，(p + 2)意味着 p 的位置之后的第二个位置，无论 p 当前指向什么位置。

例如，假如用以下代码替换前述的图 18.2 中的代码(新增了 p++)：

```
vector<char>::iterator p = container.begin();
p++;
cout << "The third entry is " << container[2] << endl;
cout << "The third entry is " << p[2] << endl;
```

```
cout << "The third entry is " << *(p + 2) << endl;
```

输出就从以下内容：

```
The third entry is C
The third entry is C
The third entry is C
```

变成以下内容：

```
The third entry is C
The third entry is D
The third entry is D
```

p++使 p 从位置 0 移至位置 1，所以 (p + 2) 现在成了指向位置 3 的迭代器，而不是指向位置 2。所以，*(p + 2) 和 p[2]等价于 container[3]，而不是 container[2]。

我们现在已充分了解了迭代器，接着可以更深刻地体会迭代器的分类方式。主要的迭代器类别如下所示：

- **正向迭代器**：支持++
- **双向迭代器**：支持++和--
- **随机访问迭代器**：支持++，--和随机访问

注意，上述类别按顺序越来越强：随机访问迭代器同时是双向迭代器；双向迭代器同时是正向迭代器。以后会讲到，不同模板容器类具有不同种类的迭代器。vector 模板类的迭代器是随机访问迭代器。

注意，正向、双向和随机访问迭代器是指迭代器的种类，不是类型名称。实际类型名称是像 std::vector<*int*>::iterator 这样的东西，它在本例中恰好是随机访问迭代器。

迭代器的种类

不同容器具有不同种类的迭代器。下面是最主要的几种：

- **正向迭代器**：支持++
- **双向迭代器**：支持++和--
- **随机访问迭代器**：支持++，--和随机访问

√ 自测题

4. 假定向量 v 按顺序包含字母'A'，'B'，'C'和'D'。以下代码输出什么？

```
vector<char>::iterator i = v.begin();
i++;
cout << *(i + 2) << " ";
i--;
cout << i[2] << " ";
cout << *(i + 2) << " ";
```

常量和可变迭代器

正向、双向和随机访问迭代器各自又划分为两个子类别：常量和可变，具体取决于提领操作符之于迭代器的行为。使用**常量迭代器**(constant iterator)，提领操作符会生成元素的

只读版本。以常量迭代器 p 为例，可将*p 赋给变量或输出到屏幕。但不能通过向*p 赋值来更改容器中的元素。相反，如 p 是**可变迭代器**(mutable iterator)，就可向*p 赋值来更改容器中的对应元素。向量迭代器是可变的，图 18.1 的以下代码对此进行了演示：

```
cout << "Setting entries to 0:\n";
for (p = container.begin(); p != container.end(); p++)
    *p = 0;
```

容器如果只有常量迭代器，就无法获得可变迭代器。但如果有可变迭代器，就可以获得容器的常量迭代器。如确定代码不会更改容器中的元素，就可能想要一个常量迭代器，并把它用作错误检查机制。例如，以下代码为名为 container 的向量容器生成常量迭代器：

```
std::vector<char>::const_iterator p = container.begin();
```

也可以使用以下等价的代码：

```
using std::vector<char>::const_iterator;
const_iterator p = container.begin();
```

像这样声明 p，以下代码就会生成错误消息：

```
*p = 'Z';
```

例如，在图 18.2 中，你可以将以下代码：

```
vector<int>::iterator p;
```

更改为以下形式：

```
vector<int>::const_iterator p;
```

这个改动不会改变图 18.2 的行为。但在图 18.1 中进行类似的更改则是不允许的，因为图 18.1 包含下面这一行代码：

```
*p = 0;
```

注意，const_iterator 是类型名称。const_iterator 类型的所有迭代器都是常量迭代器。

常量迭代器

常量迭代器是不允许更改迭代器指向的元素的一种迭代器。

逆向迭代器

有时希望逆序遍历容器中的元素。如容器有双向迭代器，可能会写出以下代码：

```
vector<int>::iterator p;
for (p = container.end(); p != container.begin(); p--)
    cout << *p << " ";
```

能成功通过编译，在某些系统上也确实能获得你想要的结果。但它存在一个原则性错误：container.end()不是普通迭代器，而是哨兵值；container.begin()则不是哨兵值。

幸好有一个简单的方法能满足要求。对于有双向迭代器的容器，可用所谓的**逆向迭代器**反转一切。以下代码能很好地工作：

```
vector<int>::reverse_iterator rp;
for (rp = container.rbegin(); rp != container.rend(); rp++)
   cout << *rp << " ";
```

成员函数 rbegin()返回定位到最后一个元素的迭代器。成员函数 rend()返回哨兵值，标志在相反顺序上元素的“末尾”。注意，对于 reverse_iterator 类型的迭代器，递增操作符++会在元素中后移；换言之，--和++的含义颠倒了。图 18.3 的程序演示了逆向迭代器。

reverse_iterator 类型还有一个常量版本，名为 const_reverse_iterator。

逆向迭代器

只要容器有双向迭代器，就可用逆向迭代器遍历容器中的所有元素。常规形式如下：

```
STL_Container<type>::reverse_iterator rp;
for (rp =c.rbegin(); rp !=c.rend(); rp++)
    在位置 rp 采取的操作;
```

其中，对象 c 的类型必须是有双向迭代器的容器类。

图 18.3　逆向迭代器

```
1  // 本程序演示了一个逆向迭代器
2  #include <iostream>
3  #include <vector>
4  using std::cout;
5  using std::endl;
6  using std::vector;
7
8
9  int main()
10 {
11     vector<char> container;
12     container.push_back('A');
13     container.push_back('B');
14     container.push_back('C');
15     cout << "Forward:\n";
16     vector<char>::iterator p;
17     for (p = container.begin(); p != container.end(); p++)
18         cout << *p << " ";
19     cout << endl;
20     cout << "Reverse:\n";
21     vector<char>::reverse_iterator rp;
22     for (rp = container.rbegin(); rp != container.rend(); rp++)
23        cout << *rp << " ";
25     cout << endl;
26     return 0;
27 }
```

示范对话

```
Forward:
A B C
Reverse:
C B A
```

其他种类的迭代器

还有另一些种类的迭代器，但本书不准备详细讨论。这里只是简单提一下你或许听说过的两种迭代器。**输入迭代器**本质上是正向迭代器，但它能用于输入流。**输出迭代器**本质上也是正向迭代器，但它能用于输出流。详情请参考讲述更高级主题的其他书籍。

√ 自测题

5. 假定向量 v 按顺序包含字母'A'，'B'，'C'和'D'。以下代码输出什么？

```
vector<char>::reverse_iterator i = v.rbegin();
i++;
i++;
cout << *i << " ";
i--;
cout << *i << " ";
```

6. 假定要运行以下代码，其中 v 是由 int 构成的向量：

```
for (p = v.begin(); p != v.end(); p++)
     cout << *p << " ";
```

可以使用以下什么方式来声明 p？

```
std::vector<int>::iterator p;
std::vector<int>::const_iterator p;
```

18.2　容　　器

把所有鸡蛋都放到一个篮子里，然后看好那个篮子。

——马克·吐温[①] (1835—1910)，《傻瓜威尔逊》

STL **容器类**是各种用于容纳数据的数据结构，比如列表、队列和栈。每种数据结构都是模板类，有一个参数指定存储什么类型的数据。例如，可指定一个列表容纳 int、double、string 或其他类/struct 类型。每个容器模板类都可以有自己的专用取值和赋值函数，以便在容器中添加和删除数据。不同容器类可能有不同种类的迭代器。例如，一个容器类可能有双向迭代器，另一个则可能只有正向迭代器。然而，所有 STL 容器类在定义好之后，迭代器操作符和成员函数 begin()及 end()都具有相同含义。

顺序容器

顺序容器用列表组织数据，列表有第一个元素、第二个元素……直至最后一个元素。第 13 章讨论的链表就是这种列表的例子。第 13 章讨论的是**单链表**(singly linked list)，因为从一个位置到另一个位置只存在一个链接。STL 没有和这种单链表对应的容器，虽然某些实现确实提供了单链表(通常命名为 slist)。STL 最简单的列表是**双链表**(doubly linked list)，也就是名为 list 的模板类。图 18.4 展示了这两种列表的区别。

---

① 马克·吐温(Mark Twain)，原名塞姆·朗赫恩·克莱门斯(Samuel Langhorne Clemens)，美国幽默大师、小说家、作家、著名演说家。——译注

图 18.4 的列表按顺序包含三个整数值：1，2 和 3。两个列表的类型分别是 slist<int> 和 list<int>。还图示了迭代器 begin() 和 end() 的位置。我们还没有讲过怎样将整数输入列表。

图 18.4 按照第 13 章的讨论，使用节点和指针来描绘单链表和双链表。STL 类 list 和非标准类 slist 可能是、也可能不是这样实现的。但使用 STL 模板类时不必关心实现细节。所以，完全可以想象列表就是由数据位置(可能是、也可能不是节点)以及迭代器(而不是指针)构成的。可将图 18.4 的箭头想象成++和--的方向 (++向下，--向上)。

介绍模板类 slist 的目的是帮你理解顺序容器。它对应第 13 章讨论的内容，也是在提到“链表”时，大多数程序员脑海中首先想到的东西。但由于模板类 slist 非标准，所以不准备继续讨论。如果你的实现提供了模板类 slist，而且想使用它，那么它的细节和后面要讲到的 list 相似，只是没有为 slist 定义递减操作符--(前缀和后缀)。

图 18.5 展示了使用 STL 模板类 list 的简单程序。push_back 函数在列表尾添加元素。注意，对于 list 模板类，使用提领操作符可读写数据。另外，对于 list 模板类以及 STL 的其他所有模板类和迭代器，所有定义都在 std 命名空间中。

图 18.4 两种列表

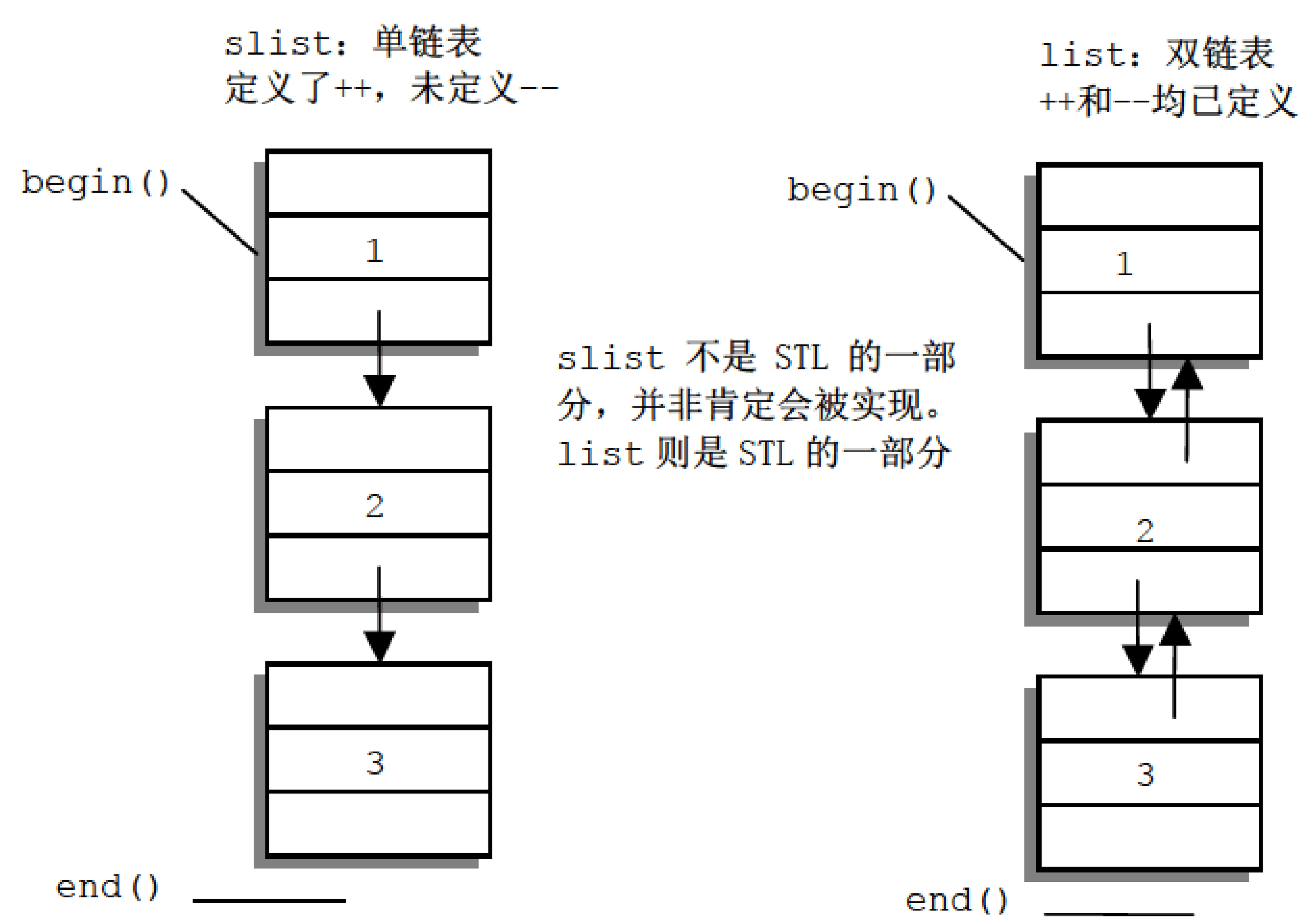


图 18.5 使用 list 模板类

```
1  //本程序演示说明 STL 模板类 list
2  #include <iostream>
3  #include <list>
4  using std::cout;
5  using std::endl;
6  using std::list;
7
8
9  int main()
10 {
11     list<int> listObject;
12
13     for (int i = 1; i <= 3; i++)
14         listObject.push_back(i);
15
16     cout << "List contains:\n";
```

```
17      list<int>::iterator iter;
18      for (iter = listObject.begin(); iter != listObject.end(); iter++)
19          cout << *iter << " ";
20
21      cout << endl;
22      cout << "Setting all entries to 0:\n";
23      for (iter = listObject.begin(); iter != listObject.end(); iter++)
24          *iter = 0;
25
26      cout << "List now contains:\n";
27      for (iter = listObject.begin(); iter != listObject.end(); iter++)
28          cout << *iter << " ";
29      cout << endl;
30
31      return 0;
32  }
```

示范对话

```
List contains:
1 2 3
Setting all entries to 0:
List now contains:
0 0 0
```

注意，将 list 和 list<int>分别替换成 vector 和 vector<int>，图 18.5 的程序一样能编译，而且结果一样。这种用法的一致性是 STL 语法的关键。

但向量和列表容器存在区别。一个主要区别是，向量容器有随机访问迭代器，而列表只有双向迭代器。例如，图 18.2 的程序使用的是随机访问。将其中所有 vector 和 vector<char>都分别替换成 list 和 list<char>，编译程序就会出错(即使删除包含 container[i]或 container[2]的语句，也会出现错误消息)。

图 18.6 总结了 STL 基本的顺序容器模板类。图 18.7 展示了一些成员函数的例子。包括栈和队列在内的其他容器，可利用 18.2 节讨论的技术从这些基本模板类获得。所有这些顺序模板类都有一个析构函数供回收内存。

图 18.6　STL 基本顺序容器

模板类名	迭代器类型名	迭代器种类	库的头文件
slist(警告: slist 不是 STL 的一部分)	slist<T>::iterator	可变正向	<slist>
	slist<T>::const_iterator	常量正向	依赖实现，也许不可用
list	list<T>::iterator	可变双向	<list>
	list<T>::const_iterator	常量双向	
	list<T>::reverse_iterator	可变双向	
	list<T>::const_reverse_iterator	常量双向	
vector	vector<T>::iterator	可变随机访问	<vector>
	vector<T>::const_iterator	常量随机访问	
	vector<T>::reverse_iterator	可变随机访问	
	vector<T>::const_reverse_iterator	常量随机访问	
deque	deque<T>::iterator	可变随机访问	<deque>
	deque<T>::const_iterator	常量随机访问	
	deque<T>::reverse_iterator	可变随机访问	
	deque<T>::const_reverse_iterator	常量随机访问	

图 18.7　一些顺序容器成员函数

成员函数(c 是容器对象)	含义
c.size()	返回容器中的元素数量
c.begin()	返回一个迭代器，定位到容器第一个元素
c.end()	返回一个迭代器，定位到容器最后一个元素之后的位置
c.rbegin()	返回一个迭代器，定位到容器最后一个元素。与 reverse_iterator 配合使用。不是 slist 的成员
c.rend()	返回一个迭代器，定位到容器第一个元素之前的位置。与 reverse_iterator 配合使用。不是 slist 的一个成员
c.push_back(*元素*)	将“*元素*”插入序列尾。不是 slist 的成员
c.push_front(*元素*)	将“*元素*”插入序列首。不是 vector 的成员
c.insert(*迭代器*, *元素*)	在“*迭代器*”的位置之前插入“*元素*”的一个拷贝
c.erase(*迭代器*)	删除“*迭代器*”位置的元素。返回新迭代器，定位到删除点之后的位置。如删除最后一个元素，就返回 c.end()
c.clear()	一个 void 函数，删除容器中的所有元素
c.front()	返回对序列首元素的引用。等价于*(c.begin())
c1 == c2	假如 c1.size() == c2.size()，而且 c1 的每个元素都等于 c2 的对应元素，结果就为 true
c1 != c2	!(c1 == c2)
<本节讨论的所有顺序容器都有一个默认构造函数、一个拷贝构造函数以及其他多种构造函数，以便将容器初始化成默认或指定元素。每个构造函数都有一个析构函数(释放并回收内存)以及一个具有良好行为的赋值操作符>	

Deque 发音为“d-queue”或“deck”，代表“Double ended queue”(双端队列)。Deque 是“超队列”。使用队列，可在队列的一端添加数据，在另一端删除数据。使用 Deque，则可在任何一端添加数据，并可在任何一端删除数据。模板类 deque 正是为 Deque 设计的模板类，它通过一个参数指定要存储的数据类型。

> **顺序容器**
>
> 顺序容器用列表排列它的数据，所以有第一个元素、第二个元素……以此类推。已讨论过的顺序容器模板类包括 slist，list，vector 和 deque。

陷阱：迭代器和删除元素

在容器中增删元素可能影响其他迭代器。通常不保证在进行了一次添加或删除操作后，迭代器还定位相同的元素。但有的容器保证迭代器不会因为添加或删除而移动(当然，除非迭代器定位的就是要删除的元素)。

在前面讲过的所有模板类中，list 和 slist 保证其迭代器不会因添加或删除而发生移动，除非迭代器定位的就是要删除的元素。但模板类 vector 和 deque 无此保证。　■

编程提示：容器中的类型定义

STL 容器类包含一些类型定义以简化编程。通过之前的学习，已知 STL 容器类可能包含 iterator，const_iterator，reverse_iterator 和 const_reverse_iterator 等类型名称(所以肯定还包含其类型定义)。此外，一般还有其他类型定义。

目前讨论的所有模板类都包含已定义类型 value_type 和 size_type。其中，value_type 是容器中存储的元素的类型。例如，list<int>::value_type 是 int 的另一个名称。size_type 是无符号整数类型，是成员函数 size 的返回类型。如第 8 章所述，向量模板类的 size_type 是 unsigned int，但完全可将该类型想象成简单 int 类型，大多数编译器不会对此表示异议。 ■

√ 自测题

7. vector 和 list 的主要区别是什么？
8. 在模板类 slist，list，vector 和 deque 中，哪些有成员函数 push_back？
9. 在模板类 slist，list，vector 和 deque 中，哪些有随机访问迭代器？
10. 在模板类 slist，list，vector 和 deque 中，哪些有可变迭代器？

容器配接器 stack 和 queue

容器配接器(container adapters)是在其他类顶部实现的模板类。例如，stack 模板类默认在 deque 模板类顶部实现。这意味着，在 stack 的实现中，“深埋”于底部的是一个 deque，后者是所有数据驻留的地方。但可忽视这个实现细节，将栈视为简单的后入/先出数据结构。

其他容器配接器还有 queue 和 priority_queue 模板类。栈和队列已在第 13 章讨论。**优先级队列(priority queue)**与普通队列相似，但有额外属性，每个数据项在入队时都被指派一个优先级。如所有数据项具有相同优先级，数据项从优先级队列中删除的方式和从普通队列中删除一样。如数据项具有不同优先级，高优先级的项将先于低优先级的项删除。我们不准备详细讨论优先级队列。之所以提到它，只是考虑到某些人可能熟悉这一概念。

虽然配接器模板类有一个默认基础容器类，但也可指定一个不同的基础容器(也许是出于对效率或其他原因的考虑，具体取决于应用)。例如，任何顺序容器都可作为 stack 的基础容器，而且除了 vector 之外的所有顺序容器都可作为 queue 的基础容器。stack 和 queue 的默认基础数据结构是 deque。priority_queue 的默认基础容器是 vector。如果你对默认基础容器类型感到满意，那么容器配接器和其他任何模板容器类都没有区别。例如，使用默认基础容器的 stack 模板类的类型名称是 stack<int>(对一个由 int 构成的栈而言)。如果希望将基础容器指定为 vector 模板类，则可以使用 stack<int, vector<int>>作为类型名称。本书总是使用默认基础容器。

如确实指定了基础容器，而且 C++的版本低于 C++11，那么记住在类型表达式中，两个>符号之间一定要留出空格，否则无法编译。例如，要写成 stack<int, vector<int> >，最后两个>之间要有空格。不要写成 stack<int, vector<int>>。C++11 和更高版本无此

要求。

图 18.8 展示了 stack 模板类的成员函数和其他细节。

图 18.8　stack 模板类

stack 配接器模板类的细节

类型名称 stack<T>或 stack<T, Underlying_Container>，表示由 T 类型的元素构成的栈。

库的头文件：<stack>，将定义放到 std 命名空间

已定义类型：value_type，size_type

无迭代器

示范成员函数

成员函数(s 是 Stack 对象)	含义
s.size()	返回栈中的元素数量
s.empty()	栈为空返回 true；否则返回 false
s.top()	返回对栈顶部“元素”的可变(可更改)引用
s.push(*元素*)	在栈顶部插入“*元素*”的拷贝
s.pop()	删除栈顶元素。注意，pop 是 void 函数。不返回被删除的元素
s1 == s2	如果 s1.size() == s2.size()，而且 s1 的每个元素都等于 s2 的对应元素，就返回 true；否则返回 false

<stack 模板类还有一个默认构造函数、一个拷贝构造函数以及一个特殊的构造函数(获取任何顺序容器类的一个对象，并用序列中的元素来初始化栈)。此外，还有一个析构函数(释放并回收内存)以及一个具有良好行为的赋值操作符>

queue 模板类的细节则在图 18.9 中展示。

图 18.9　queue 模板类

queue 配接器模板类的细节

类型名称：queue<T>或 queue<T, Underlying_Container>，表示由 T 类型的元素构成的一个队列

为效率起见，Underlying_Container 不能是 vector 类型

库的头文件：<queue>，将定义放到 std 命名空间

已定义类型：value_type，size_type

无迭代器

示范成员函数

成员函数(q 是 Queue 对象)	含义
q.size()	返回队列中的元素数量
q.empty()	队列为空返回 true；否则返回 false
q.front()	返回对队列首元素的一个可变引用
q.back ()	返回对队列尾元素的一个可变引用
q.push(*元素*)	将“*元素*”添加到队列尾
q.pop()	删除栈的首元素。注意，pop 是 void 函数，不返回被删除的元素
q1 == q2	如果 q1.size() == q2.size()，而且 q1 的每个元素都等于 q2 中对应的元素，就返回 true；否则返回 false

<queue 模板类还有一个默认构造函数、一个拷贝构造函数以及一个特殊的构造函数(获取任何顺序容器类的一个对象，并用序列中的元素初始化队列)。此外，还有一个析构函数(释放内存以便回收)以及一个具有良好行为的赋值操作符>

图 18.10 是使用 stack 模板类的简单例子。

图 18.10　使用 Stack 模板类的程序

```
1  // 本程序演示了 STL stack 模板类的用法
2  #include <iostream>
3  #include <stack>
4  using std::cin;
5  using std::cout;
6  using std::endl;
7  using std::stack;
8
9  int main()
10 {
11     stack<char> s;
12
13     cout << "Enter a line of text:\n";
14     char next;
15     cin.get(next);
16     while (next != '\n')
17     {
18         s.push(next);
19         cin.get(next);
20     }
21
22     cout << "Written backward that is:\n";
23     while ( ! s.empty() )
24     {
25         cout << s.top();
26         s.pop();
27     }
28     cout << endl;
29
30     return 0;
31 }
```

成员函数 pop 移除一个元素，但不返回那个元素。pop 是 void 函数。所以，要用 top 读取移除的元素

示范对话

```
Enter a line of text:
straw
Written backward that is:
warts
```

√ 自测题

11. stack 模板配接器类有哪些种类的迭代器(正向、双向或随机访问)?

12. queue 模板配接器类有哪些种类的迭代器(正向、双向或随机访问)?

13. 假如 s 是一个 stack<***char***>，s.pop()的返回值类型是什么?

关联容器 set 和 map

关联容器(associative container)是一种非常简单的数据库。它们负责存储数据，比如 struct 或其他类型的数据。每个数据项都有一个关联的值，称为该数据项的**键**(key)。例如，假定数据项是一个 struct，其中包含一条员工记录，那么相应的键就可能是员工的社会保险号(或身份证号)。数据项通过键来检索。键的类型和要存储的数据的类型不一定有任何

关系，虽然它们通常都是相关的。一种非常简单的情况是每个数据项同时也是它自己的键。例如，在一个 set 中，每个元素都是它自己的键。

从某种意义上说，set 模板类是最简单的容器。它存储不重复的元素。一个元素在插入 set 时，只有第一次插入操作才有效，之后的其他插入都无效，所以不会有重复的元素。每个元素都是它自己的键。你只需添加或删除元素，并查询一个元素是否在 set 中。和所有 STL 类一样，set 模板类在编写时以效率为目标。为了高效工作，set 对象按排序顺序存储它的值。可像下面这样指定元素的存储顺序：

```
set<T, Ordering> s;
```

其中，*Ordering*[1]应该是一个具有良好行为的排序关系，它获取 T 类型的两个参数，返回一个 bool 值。T 是存储的元素的类型。未指定 *Ordering*，就假定 *Ordering* 是<关系操作符。图 18.11 总结了 set 模板类的基本细节。

图 18.11　set 模板类

set 模板类的细节

类型名称：set<T>或 set<T, Ordering>，表示由 T 类型的元素构成的一个 set

Ordering 用于对元素进行排序，以便存储。未指定 Ordering，就假定 Ordering 是二元操作符<

库的头文件：<set>，将定义放到 std 命名空间

已定义类型包括：value_type，size_type

迭代器：iterator，const_iterator，reverse_iterator 以及 const_reverse_iterator

所有迭代器都是双向的，而且除了含有 const_前缀的之外，其他都是可变(可修改)的。

begin()，end()，rbegin()和 rend()具有预期的行为。

增删元素不影响迭代器，除非定位的是要删除的元素。

示范成员函数

成员函数(s 是 Set 对象)	含义
s.insert(*元素*)	将“*元素*”的一个拷贝插入 set。如果 set 中已经有这个“元素”，此次插入操作没有任何效果
s.erase(*元素*)	从 set 中删除“*元素*”。如果“*元素*”不在 set 中，此次删除操作没有任何效果
s.find(*元素*)	返回一个可变迭代器，它定位到 set 中的*元素*的一个拷贝。如果“*元素*”不在 set 中，就返回 s.end()
s.erase(*迭代器*)	删除由“*迭代器*”定位的位置处的元素
s.size()	返回 set 中的元素数量
s.empty()	set 为空返回 true；否则返回 false
s1 == s2	如果两个 set 包含相同的元素，就返回 true；否则返回 false

set 模板类还有一个默认构造函数、一个拷贝构造函数以及这里没有列出的其他特殊构造函数。此外，还有一个析构函数(返回占用的内存以便回收)以及一个具有良好行为的赋值操作符

图 18.12 用一个简单例子展示了如何使用模板类 set 的部分成员函数。

[1] Ordering 必须是一个 strict weak ordering (严格的弱序化)。用于实现<操作符的大多数典型的 Ordering 都是 strict weak ordering。想要了解细节的读者注意，作为 strict weak ordering，Ordering(x, x)肯定为 false；Ordering(x, y)暗示着!Ordering(y, x)；Ordering(x, y)和 Ordering(y, z)暗示着 Ordering(x, z)；假如 x 等价于 y，而且 y 等价于 z，那么 x 等价于z。假如 Ordering(x, y)和 Ordering(y, x)都为 false，那么两个元素 x 和 y 就是等价的。

图 18.12　使用 set 模板类的程序

```
1  // 本程序演示说明 set 模板类的用法
2  #include <iostream>
3  #include <set>
4  using std::cout;
5  using std::endl;
6  using std::set;
7
8  int main()
9  {
10     set<char> s;
11
12     s.insert('A');
13     s.insert('D');
14     s.insert('D');     <—— 不管将一个元素加入一个 set 多少次，set
15     s.insert('C');          都只包含那个元素的一个拷贝
16     s.insert('C');
17     s.insert('B');
18
19     cout << "The set contains:\n";
20     set<char>::const_iterator p;
21     for (p = s.begin(); p != s.end(); p++)
22        cout << *p << " ";
23     cout << endl;
24
25     cout << "Removing C.\n";
26     s.erase('C');
27     for (p = s.begin(); p != s.end(); p++)
28        cout << *p << " ";
29     cout << endl;
30
31    return 0;
32 }
```

示范对话

```
The set contains:
A B C D
Removing C.
A B D
```

映射(map)是排好序的“键/值”对的容器。每个“键”都对应一个(而且只有一个)“值”，从而建立像(*键, 值*)这样的映射关系。STL 模板类 map 实现了映射。例如，要为每个字符串名称都分配不重复的编号，可像下面这样声明 map 对象：

```
map<string, int> numberMap;
```

numberMap 对象可为 string 值(“键”)关联不重复的 int 值。

还可将映射想象成**关联数组**(associative array)。传统数组是从一个数值索引映射到一个值。例如，a[10] = 5 将数字 5 存储到索引位置 10 处。关联数组则允许用自选数据类型定义索引。例如，numberMap["c++"] = 5 将整数 5 与字符串"c++"关联。为方便起见，定义了[]方括号操作符以便使用数组风格的表示法访问映射——虽然还可使用 insert 或 find 方法。

和 set 对象相似，map 对象根据键来排序并存储元素。尖括号<>中可指定第三项，也就是键的 Ordering。不指定就用默认 Ordering。具体能用的 Ordering 和 set 模板类允许使用的 Ordering 一样。注意 Ordering 只适用于键。第二种类型可为任何类型，而且不必和任

何 Ordering 有任何联系。和 set 对象一样，之所以要对 map 对象中存储的条目进行排序，是出于对效率的考虑。

为了在 map 中添加和获取数据，最简单的办法就是使用[]操作符。给定 map 对象 m，表达式 m[key]返回对 key 关联数据元素的引用。如 map 中不存在和 key 关联的项，就创建新项，并赋予其数据元素的默认值。对于数值数据类型，默认值是 0。对于 string 类型的对象，默认值是空字符串。

[]操作符可用于在 map 中添加新项，也可用于替换现有项。例如，m[key] = newData;这个语句会在 key 和 newData 之间创建新的关联。在这个过程中必须小心，避免错误地创建映射项。例如，假定执行 val = m[key];这个语句，并希望获取与 key 关联的值，但错误地输入了 map 中不存在的一个 key 值，就会为你输入的 key 创建新项，并将新项默认值赋给 val。

图 18.13 展示了 map 模板类的基本细节。为了理解这些细节，需要先知道 pair 模板类的一些事情。

图 18.13　map 模板类

map 模板类的细节

类型名称：map<KeyType, T>或者 map<KeyType, T, Ordering>，针对一个 map，它将 KeyType 类型的元素关联(映射)到 T 类型的元素。

Ordering 根据键对元素进行排序，以实现高效率存储。没有指定 Ordering 就使用默认 Ordering，即二元操作符<。

库的头文件：<map>，它将定义放到 std 命名空间中。

已定义类型包括：key_type 表示键的类型，mapped_type 表示键映射到的值的类型，另外还有 size_type(已定义类型 key_type 就是前面说的 KeyType)。

迭代器：iterator，const_iterator，reverse_iterator 以及 const_reverse_iterator。所有迭代器都是双向的。除了以 const_开头的之外，所有迭代器既非常量迭代器，也非可变迭代器，而是介于两者之间。例如，假定 p 具有 iterator 类型，那么可以更改键，但不可更改 T 类型的值。初学者最好将所有迭代器都视为常量迭代器。

begin()，end()，rbegin()和 rend()具有预期的行为。

增删元素不会影响迭代器，除非迭代器正好定位到要删除的元素。

示范成员函数

成员函数(m 是 Map 对象)	含义
m.insert(*元素*)	在 map 中插入*元素*，*元素*具有 pair<KeyType, T>类型。返回 pair<iterator, bool>类型的值。假如插入成功，返回的 pair 的第二个部分就为 true，而且 iterator(迭代器)定位到刚才插入的元素
m.erase(Target_Key)	删除键为 Target_Key 的元素
m.find(Target_Key)	返回迭代器，定位到键为 Target_Key 的元素。如果没有找到匹配元素，就返回 m.end()
m.size()	返回 map 中的键/值对的数量
m.empty()	map 为空返回 true；否则返回 false
m1 == m2	两个 map 包含相同的键/值对返回 true；否则返回 false。

map 模板类还有一个默认构造函数、一个拷贝构造函数以及其他未在这里提及的构造函数。此外，还有一个析构函数(释放内存以便回收)以及一个具有良好行为的赋值操作符

STL 模板类 pair<T1, T2>包含值对构成的对象。在一对值中，第一个元素具有类型 T1，第二个具有类型 T2。假定 aPair 是 pair<T1, T2>类型的对象，那么 aPair.first 是第一个元素，类型是 T1。aPair.second 是第二个元素，类型是 T2。成员变量 first 和 second 是公共成员变量，所以不需要取值或赋值函数。

pair 模板的头文件是<utility>。所以为了使用 pair 模板类，要在源文件中包含以下语句(或其他类似语句)：

```
#include<utility>
using std::pair;
```

map 模板类用 pair 模板类存储 key 与一个数据项之间的关联。例如，给定以下定义：

```
map<string, int> numberMap;
```

可以使用 pair 对象添加从"c++"到数字 10 的映射：

```
pair<string, int> toInsert("c++", 10);
numberMap.insert(toInsert);
```

也可以使用 [] 操作符：

```
numberMap["c++"] = 10;
```

任何情况下用迭代器访问该 pair，iterator->first 引用键"c++"，iterator->second 引用数据值 10。图 18.14 演示了如何使用 map 模板类的一些成员函数。

图 18.14　使用 map 模板类的程序

```
1  //该程序演示了 map 模板类的用法
2  #include <iostream>
3  #include <map>
4  #include <string>
5  using std::cout;
6  using std::endl;
7  using std::map;
8  using std::string;
9
10 int main()
11 {
12   map<string, string> planets;
13
14   planets["Mercury"] = "Hot planet";
15   planets["Venus"] = "Atmosphere of sulfuric acid";
16   planets["Earth"] = "Home";
17   planets["Mars"] = "The Red Planet";
18   planets["Jupiter"] = "Largest planet in our solar system";
19   planets["Saturn"] = "Has rings";
20   planets["Uranus"] = "Tilts on its side";
21   planets["Neptune"] = "1500 mile-per-hour winds";
22   planets["Pluto"] = "Dwarf planet";
23
24   cout << "Entry for Mercury - " << planets["Mercury"]
25          << endl << endl;
26
27   if (planets.find("Mercury") != planets.end())
28      cout << "Mercury is in the map." << endl;
29   if (planets.find("Ceres") == planets.end())
```

```
30        cout << "Ceres is not in the map." << endl << endl;
31
32     cout << "Iterating through all planets: " << endl;
33     map<string, string>::const_iterator iter;
34     for (iter = planets.begin(); iter != planets.end(); iter++)
35     {
36        cout << iter->first << " - " << iter->second << endl;
37     }
38     return 0;
39 }
```

示范对话

```
Entry for Mercury - Hot planet

Mercury is in the map.
Ceres is not in the map.

Iterating through all planets:
Earth - Home
Jupiter - Largest planet in our solar system
Mars - The Red Planet
Mercury - Hot planet
Neptune - 1500 mile-per-hour winds
Pluto - Dwarf planet
Saturn - Has rings
Uranus - Tilts on its side
Venus - Atmosphere of sulfuric acid
```

迭代器输出 map 时，将按 key 排序。本例按行星名称首字母顺序排序

虽然不打算讲述细节，但还是要提一下另外两种关联容器。模板类 multiset 和 multimap 在本质上分别等同于 set 和 map，只是 multiset 允许存储重复的元素，而 multimap 允许为每个键关联多个值。

编程提示：为容器使用初始化、基于范围的 for 和 auto

C++11 引入了几项新功能来简化集合编程。具体地说，可用统一的初始化列表格式来初始化容器。初始化数据放到一对花括号中，即为**初始化列表**。还可使用 auto 和基于范围的 for 循环来轻松遍历容器。

例如以下初始化好的集合对象：

```
map<int, string> personIDs = {
                                {1,"Walt"},
                                {2,"Kenrick"}
                              };
set<string> colors = {"red","green","blue"};
```

可使用基于范围的 for 循环和 auto 来简单地遍历每个容器：

```
for (auto p : personIDs)
   cout << p.first << " " << p.second << endl;
for (auto p : colors)
   cout << p << " ";
```

输出如下所示：

```
1 Walt
2 Kenrick
blue green red
```

■

视频讲解：*C++11 and Containers*

效率问题

STL 在设计时将效率作为重要的考虑因素。事实上，STL 的各种实现都在力求获得最优效率。例如，set 和 map 中的元素按排序顺序存储，使搜索算法更高效。

每个模板类的每个成员函数都保证了一个最大运行时间。这些最大运行时间是用大 *O* 表示法来表示的，详情在下一节讨论(会列出前面讨论的一些容器成员函数的保证运行时间。)一些更高级的参考书(甚至本章后文)会讲到特定函数的保证最大运行时间。

自测题

14. 执行以下代码之后，mymap 中有多少个元素？

```
map<int, string> mymap;
mymap[5] = "c++";
cout << mymap[4] << endl;
```

15. 一个 set 可以包含类类型的元素吗？

16. 假定 s 具有 set<*char*>类型，而且'A'在 s 中，s.find('A')返回什么值？'A'不在 s 中又返回什么值？

18.3　泛 型 算 法

> 祖传秘方，包治百病，头痛发烧、精神不振、血气不足、功能紊乱、风湿腰痛、各种胃病、支气管炎、胆肾结石，药到病除！
>
> ——*江湖郎中的吆喝*

本节要讨论 STL 的一些基本函数模板。因篇幅有限，不可能对它们进行非常全面的描述。不过，将提供一个足够大的例子，使你很好地体会 STL 包含的内容，同时会提供足够多的细节，让你开始使用这些模板函数。

这些模板函数有时称为**泛型算法**(generic algorithm)。用“算法”一词是有原因的。记住，算法是完成任务的一组指令。可用任何语言描述算法，其中包括 C++这样的编程语言。但听到“算法”一词时，程序员首先想到的往往是用日常语言或伪代码进行的非正式描述。所以，通常将算法视为对函数定义代码的一种抽象。它虽然会给出一些重要细节，但不一定要细致到可直接编码的程度。STL 规定了构成 STL 模板函数基础的某些细节。正因为此，有时将它们称为“泛型算法”。

这些 STL 函数模板并非只是返回一个值那么简单。STL 中的函数模板提出了一些最起码的要求，要符合标准，实现这些函数模板时必须满足这些要求。大多数时候，都必须满足一个“保证运行时间”(guaranteed running time)。这就为函数接口的概念开辟了一个全新的思路。在 STL 中，接口不仅要告诉程序员函数能做什么和如何用，还要指出如何快速完成任务。某些时候，标准甚至规定了必须采用的算法——虽然不会给出详尽的编码细节。此外，假如规定了要采用的算法，那肯定是因为该算法的效率已得到证明。本章使用“泛

型算法”、“泛型函数”以及“STL 函数模板”描述同一种东西。

讨论这些模板函数或泛型算法的效率时，必须应该掌握一定的术语。下面先学习一些背景知识，了解平时如何对算法效率进行衡量。

运行时间和大 O 表示法

要是问程序员程序有多快，他们可能会给出像“两秒”这样的答案。但程序速度不能用单个数字来衡量。和输入量小的时候相比，输入量大时运行时间通常更长。这很容易理解，程序对 10 个数和 1000 个数进行排序，前者花的时间通常会短一些。可能花两秒对 10 个数排序，但要花 10 秒对 1000 个数排序。那么，程序员如何正确回答“程序有多快”呢？

这种情况下，程序员应提供一张表格，列出在不同输入量的时候程序所花的时间，图 18.15 展示了一个例子。表格给出的不是单个时间，而是不同输入量时的不同执行时间。可将这个表格视为函数的数学表示。就像 C++函数要获取参数并返回值，该函数也要获取参数(输入量)，并返回一个数字(程序在此输入量下的运行时间)。假定函数名为 T，那么 $T(10)$ 等于 2 秒，$T(100)$等于 2.1 秒，$T(1000)$等于 10 秒，$T(10000)$等于 2.5 分钟。表格只列出了可能传给函数 T 的一部分值。提供任何输入量，程序都要花一定时间执行。所以，虽然表格中没有显示，但 $T(1)$，$T(2)$，……，$T(101)$，$T(102)$等都有对应的值。对于任何正整数 N，$T(N)$表示的都是程序对 N 个数进行排序所花的时间。我们将函数 T 称为程序的**运行时间**。

图 18.15　“运行时间”函数的示范值

输入量	运行时间
10 个数	2 秒
100 个数	2.1 秒
1 000 个数	10 秒
10 000 个数	2.5 分钟

注意，我们假定该排序程序就 N 个数字的任何列表花费相同的运行时间。但这并不一定正确。列表已排好序，花的时间也许会少很多。这种情况下，$T(N)$被定义成“最难”列表所需的时间；换言之，同样 N 个数字的列表，该列表会使程序花最长的运行时间。这称为**最差情况运行时间**。本章为一种算法或者一些代码给出运行时间时，所指的肯定是“最差情况运行时间”。

程序或算法所花时间通常由公式给出，比如 $4N+3$，$5N+4$ 或者 N^2。假如运行时间 $T(N)$ 等于 $5N+5$，表示在输入量为 N 的前提下，程序员将运行 $5N+5$ 个时间单位。

以下代码搜索包含 N 个元素的数组 a，判断 target 值是否在数组中：

```
int i = 0;
bool found = false;
while (( i < N) && !(found))
   if (a[i] == target)
      found = true;
   else
      i++;
```

我们希望计算一些值，估算计算机花多少时间执行上述代码。这种估算值不应依赖具体所用的计算机，这是由于不知道要用哪台计算机，或者想在不同计算机上在不同时间运行程序。一种可能是统计执行步数，但很难对“步”进行量化。这种情况下的常规做法是

统计操作(operation)数。“操作”和“步”的含义其实都比较模糊，但在实际应用中，对于如何量化一个“操作”，人们起码有一些协定。假设对于 C++代码，每执行以下任何一个操作，都计为一个 operation：=，<，&&，!，[]，==和++。除了执行这些操作，计算机还必须做其他事情，但上述操作是最主要的，我们假定在代码运行时，它们将占用大多数时间。事实上，在分析时间时，会假设其他任何任务完全不费时间，程序总的运行时间就是执行上述操作的时间。虽然这个假设并不理想，而且肯定不完全符合事实，但确实简化了假设，而且能取得基本正确的结果，所以在分析程序或算法时，往往采用这一假设。

即使简化了假设，还有两种情况必须考虑：target 在数组中和不在数组中。首先假定 target 不在数组中。执行的操作数将取决于搜索的数组元素数。循环执行前，=操作执行两次。由于假定 target 不在数组中，所以循环迭代 N 次，为数组中的每个元素逐一执行一次。每次循环迭代都执行以下操作：<，&&，!，[]，==和++。所以，在 N 次循环迭代中，每次都要产生 6 次操作。最后，在 N 次迭代之后，将再次检查布尔表达式，发现它为 false。这便产生了最后三次操作(<，&&，!)[①]。统计所有这些操作，结果是当 target 不在数组中时，总操作数是 $6N + 5$。至于当 target 在数组中，总操作数是多少，将作为课后作业留给读者。请自行验证。答案应该是“$6N + 5$ 或更少”。所以，“最差情况运行时间”是：对于包含 N 个元素的任意数组，以及任何 target 值，$T(N) = 6N + 5$ 次操作。

现在，我们知道对于上述搜索代码，最差情况运行时间是 $6N + 5$ 次操作。但是，“操作”并不是传统时间单位(比如纳秒、秒或分钟)。要知道算法在特定计算机上花多长时间，就必须知道计算机执行一个操作所需的时间。如一个操作花 1 纳秒，可以说时间是 $6N + 5$ 纳秒。如一个操作花 1 秒，可以说时间是 $6N + 5$ 秒。一台很慢的计算机，一个操作要花 10 秒，时间则是 $60N + 50$ 秒。通常，如果计算机要花 c 纳秒执行一个操作，那么实际运行时间大约为 $c(6N + 5)$纳秒(之所以说“大约”，是因为进行了一些简化假定，所以结果可能不是准确的运行时间)。这意味着 $6N + 5$ 的运行时间是非常粗略的估计。为了获得以纳秒表示的运行时间，必须再乘以一个常量(取决于实际所用的计算机)。有一个标准方法表示这种约值，我们接着将讨论这种表示法。

估算运行时间时(比如刚才描述的时间)，通常采用大 O 表示法来表示(O 是字母“Oh”，不是阿拉伯数字零)。假定估计运行时间为 $6N + 5$ 个操作，并假定无论每个操作花多长时间，都肯定有一个常量乘数 c，所以实际运行时间小于或等于 $c(6N + 5)$。

这种情况下，我们说代码(或程序、算法)的运行时间是 $O(6N + 5)$，通常读作“$6N + 5$ 的大 O”。不需要知道常量 c 具体是多少。它无疑每台计算机都不同。但必须知道对于任何计算机系统，都存在这样的一个 c 值。如计算机非常快，c 可能小于 1，比如 0.001。计算机非常慢，c 可能非常大，比如 1000。此外，更改单位(比如从纳秒变成秒)只需更改常量乘数，所以不需要给出任何时间单位。

注意，大 O 估计的是上限约值。我们总是取上限值而不是下界值。还要注意，计算大 O 约值不需要获得很准确的操作计数。我们只要求估值在“乘以一个常量乘数”之后正确即可。即使估值是真实值的两倍，也不一定有错。

在 $6N + 5$ 这样的公式中，参数 N 代表算法(或程序、代码段)要解决的任务的规模。上

① 由于短路算法，!(found)不会求值，所以实际是两次操作，而不是三次操作。但这里的重点是获得一个好的上限值。在这里，添加一次额外的操作不会造成太大影响。

例的参数 N 是要搜索的数组元素数。显然，数组包含的元素越多，花的时间越长。大 O 运行时间总是表示成这样的一个函数，它以问题规模为参数。本章的所有算法都涉及某个容器中的一个范围中的值。所有情况下，N 都是那个范围中的元素个数。

下面从编程角度探讨大 O 估算：

只找具有最高指数的项，不要关心常量乘数。

例如，以下全都是 $O(N^2)$：

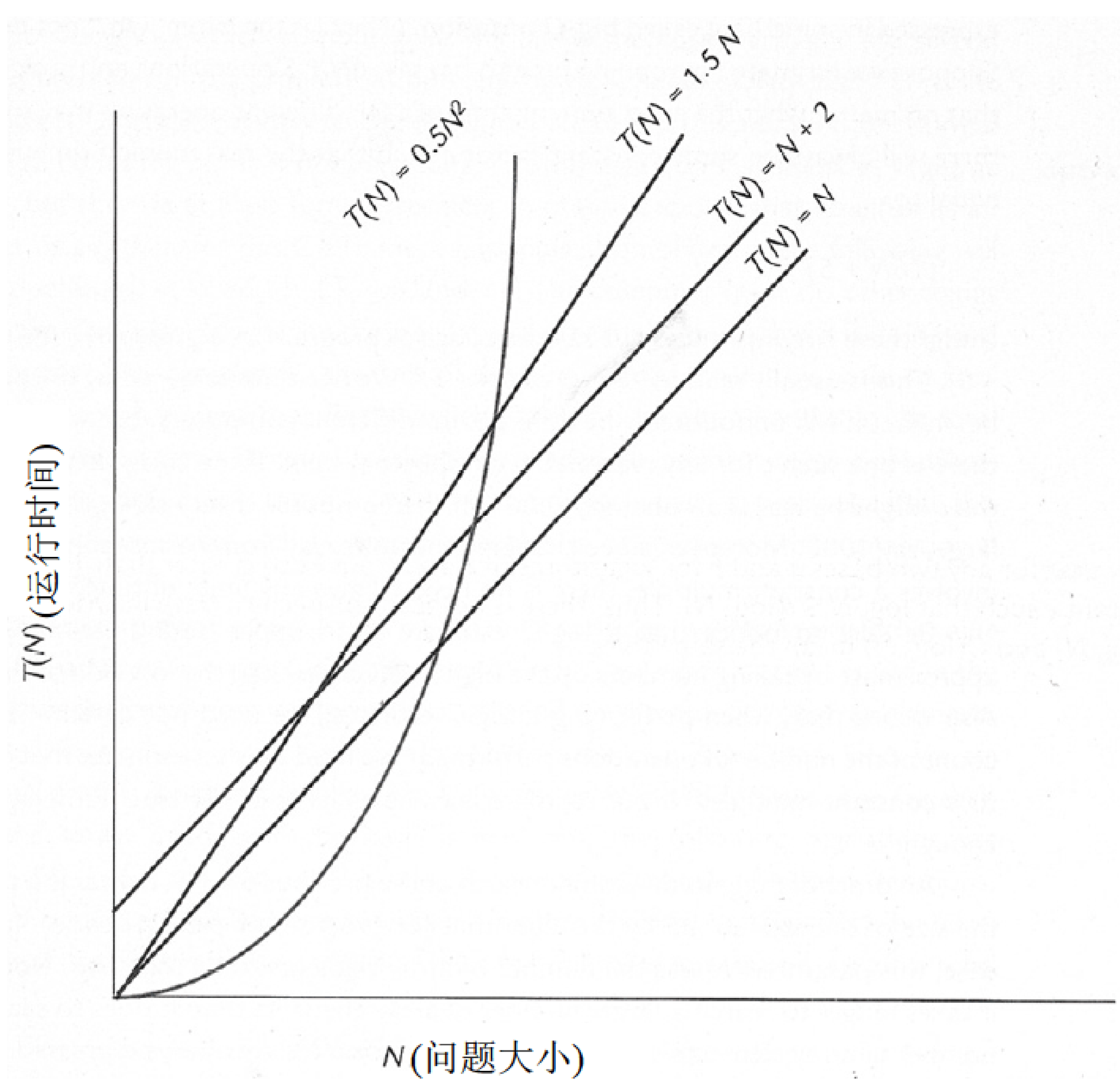
$$N^2 + 2N + 1,\quad 3N^2 + 7,\quad 100N^2 + N$$

以下则全都是 $O(N^3)$：

$$N^3 + 5N^2 + N + 1,\quad 8N^3 + 7,\ 100N^3 + 4N + 1$$

大 O 运行时间估算诚然十分粗略，但确实能从这些估值中看出一些端倪。用它无法看出运行时间 $5N + 5$ 和运行时间 $100N$ 的区别，但有时确实能区分一些运行时间，并判断某些算法快于其他算法。如图 18.16 所示，$O(N)$的所有函数的图最终都落在函数 $0.5N^2$ 的下方。结果很明显，使用足够大的 N 值，任何 $O(N)$算法最终肯定比任何 $O(N^2)$算法快。$O(N^2)$算法在解决特定规模的问题时可能快于 $O(N)$算法，但程序员发现在实际应用中，对于大多数较大规模的应用，$O(N)$算法都要优于 $O(N^2)$算法。另外两种不同的大 O 运行时间也如此。

图 18.16　比较运行时间



了解一些术语有助于理解泛型算法的运行时间。**线性运行时间**(linear running time)是指 $T(N) = aN + b$ 的运行时间。线性运行时间肯定是 $O(N)$运行时间。**二次运行时间**(quadratic running time)是指最高项为 N^2 的运行时间。二次运行时间肯定是 $O(N^2)$运行时间。偶尔也在运行时间公式中使用对数。但在列出这种公式时，一般不给出任何对数底，因为对数底只是常量乘数。看见 $\log N$，把它看成以 2 为底的 N 的对数。但看成以 10 为底的 N 的对数也无大碍。对数是一种“慢增长”(slow-growing)函数，所以 $O(\log N)$是非常快的运行时间。

有时将 $\log_2 N$ 写成 lg N。

容器访问运行时间

学习了大 O 表示法之后，就可用它表示 18.2 节讨论的容器类的一些访问函数的效率。在 vector 末尾插入(push_back)，在 deque 首尾处插入(push_back 和 push_front)，以及在 list 的任何地方插入(insert)，所有这些操作的运行时间都是 $O(1)$。换言之，运行时间有一个常量上限，不管容器本身有多大。在 vector 或 deque 中，插入或删除任意一个元素的运行时间是 $O(N)$。其中，N 是容器中的元素数。对于 set 或 map，一个查找操作(find)的运行时间是 $O(\log N)$。其中，N 是容器中的元素数。

√ 自测题

17. 证明运行时间 $T(N) = aN + b$ 是一个 $O(N)$运行时间。**提示：**唯一的问题是$+ b$。假定 N 至少为 1。
18. 证明对于对数的任意两个底 a 和 b，假如 a 和 b 都大于 1，那么存在一个常量 c，使 $\log_a N <= c(\log_b N)$。所以，没必要在 $O(\log N)$中指定一个具体的底。换言之，$O(\log_a N)$和 $O(\log_b N)$是一码事。

不修改容器的算法

本节要描述一些作用于容器、但不以任何方式修改容器内容的模板函数。一个简单和典型的例子是泛型 find。

泛型 find 函数类似于 set 模板类的 find 成员函数，但两者是不同的函数。具体地说，泛型 find 函数要求获取比讨论 set 模板类时介绍的 find 函数更多的参数。泛型 find 函数搜索容器，在其中定位特定元素，但这个泛型 find 可作用于任意 STL 顺序容器类。图 18.17 展示了为 vector<char>类使用泛型 find 函数的例子。将 vector<char>全文替换成 list<char>，或者将 vector<char>替换成其他任何顺序容器类，图 18.17 的函数将具有一样的行为。正因为此，这种函数才被称为**泛型函数**(generic function)。find 函数的一个定义适合多种容器。

假如 find 函数没有找到匹配项，就返回它的第二个迭代器实参。图 18.17 中，这个实参是 line.end()，但也可选择其他实参。示范对话 2 展示了 find 没有找到它要找的内容时的结果。

find 绝对适合任何容器类吗？答案是否定的。首先，它要获取迭代器作为参数，但某些容器(比如 stack)没有迭代器。为了使用 find 函数，容器必须有迭代器，元素必须存储成一种线性序列，这样++操作符才能移动迭代器来遍历整个容器。此外，元素还必须能用==进行比较。换言之，容器必须有正向迭代器(或增强的迭代器，比如双向迭代器)。

图 18.17　泛型函数 find

```
1  // 本程序演示了如何使用泛型函数 find
2  #include <iostream>
3  #include <vector>
4  #include <algorithm>
5  using std::cin;
6  using std::cout;
7  using std::endl;
8  using std::vector;
```

```
 9  using std::find;
10
11  int main()
12  {
13      vector<char> line;

14      cout << "Enter a line of text:\n";
15      char next;
16      cin.get(next);
17      while (next != '\n')
18      {
19          line.push_back(next);
20          cin.get(next);
21      }

22      vector<char>::const_iterator where;
23      where = find(line.begin(), line.end(), 'e');
24      //where 定位到 v 中第一个'e'所在的位置

25      vector<char>::const_iterator p;
26      cout << "You entered the following before you entered your first e:\n";
27      for (p = line.begin(); p != where; p++)
28          cout << *p;
29      cout << endl;

30      cout << "You entered the following after that:\n";
31      for (p = where; p != line.end(); p++)
32          cout << *p;
33      cout << endl;

34      cout << "End of demonstration.\n";
35      return 0;
36  }
```

如果 find 没有找到匹配项，就返回它的第二个实参

示范对话 1

```
Enter a line of text
A line of text.
You entered the following before you entered your first e:
A lin
You entered the following after that:
e of text.
End of demonstration.
```

示范对话 2

```
Enter a line of text
I will not!
You entered the following before you entered your first e:
I will not!
You entered the following after that:
End of demonstration.
```

如果 find 没有找到匹配项，就返回 line.end()

下面描述泛型函数模板时，将所需迭代器种类的名称用作类型参数名，以便描述迭代器类型参数。所以，ForwardIterator 应替换成某种正向迭代器类型，比如 list、vector 或者其他容器模板类中的 iterator 类型。记住，双向迭代器同时是正向迭代器，随机访问迭代器同时是双向迭代器。所以，类型名称 ForwardIterator 除了适合普通的正向迭代器类型，还适合任何双向或随机访问迭代器类型。指定 ForwardIterator 时，有时甚至可以使用更简单的迭代器种类，即输入迭代器或输出迭代器。但由于没有深入讨论输入和输出迭代器，所以在函数模板声明中没有提到它们。

记住，“正向迭代器”、“双向迭代器”以及“随机访问迭代器”等名称是指迭代器的种类，不是实际类型名称。实际类型名称是像 std::vector<int>::iterator 这样的形式(这个例子代表的是一个随机访问迭代器)。

图 18.18 展示了 STL 的一些不修改内容的泛型函数的例子。我们采用了讨论容器迭代器时常用的一种表示法。从迭代器 first 移至(但不等于)迭代器 last，途经的迭代器位置表示成 range[first, last)或者范围[first, last)。例如，以下 for 循环输出范围[first, last)中的所有元素：

```
for (iterator p = first; p != last; p++)
    cout << *p << endl;
```

注意，给定两个 range 时，它们不一定在同一个容器中，所在的容器甚至不一定是相同类型的。例如，对于 search 函数，[first1, last1]和[first2, last2]这两个 range 可能在相同或不同的容器中。

range [first, last)

经常要将迭代器 first 定位的位置(通常是 container.begin())移至(但不包括)last 定位的位置(通常是 container.end())。由于这个操作相当常用，所以专门为它设计了一个特殊名称，即 range[first, last)或者范围[first, last)。例如，以下代码输出范围[c.begin(), c.end())中的所有元素。其中，c 是某个容器对象，比如一个 vector：

```
for (iterator p = c.begin(); p != c.end(); p++)
    cout << *p << endl;
```

注意，图 18.18 有三个搜索函数，即 find，search 和 binary_search。其中，search 函数搜索一个子序列，而 find 和 binary_search 函数搜索单个值。那么，需要搜索单个元素时，应该使用 find 还是 binary_search 呢？一个函数返回的是迭代器，另一个则返回布尔值。但这并不是两者最大的区别。binary_search 函数要求搜索的 range 已排序(使用<按升序排列)，而且运行时间是 $O(\log N)$。相反，find 函数不要求 range 已排序，它只能保证运行时间为线性时间。所以，假如元素已排好序，或者你能事先排序，那么最好用 binary_search 来搜索，这样速度会快很多。

图 18.18　不修改容器的泛型函数

以下函数全都适合正向迭代器，所以同样适合双向和随机访问迭代器。
(某些情况下，甚至适合其他种类的迭代器，只是本章未深入讨论)。

```
1  template <class ForwardIterator, class T>
2  ForwardIterator find(ForwardIterator first,
3                       ForwardIterator last, const T& target);
4  //遍历范围[first, last)。返回定位到第一个 target 的迭代器；未找到就返回第二个实参
5  //时间复杂度：与范围[first, last]的长度成正比的线性时间
6

7  template<class ForwardIterator, class T>
8  int① count(ForwardIterator first, ForwardIterator last,const T& target);
9  //遍历 range [first, last]，返回和 target 相等的元素的数量
10 //时间复杂度：与范围[first, last]的长度成正比的线性时间
```

① 实际返回类型是我们没有讨论过的一个整数类型，但返回值应该能赋给 int 类型的变量。

```
11

12  template<class ForwardIterator1, class ForwardIterator2>
13  bool equal(ForwardIterator1 first1, ForwardIterator1 last1,
14            ForwardIterator2 first2);
15  //如果[first1, last1]包含的元素及元素的顺序与 first2 处开始的前 last1-first1 个
16  //元素及其顺序相同，就返回 true。否则，返回 false
17  //时间复杂度：与范围[first, last]的长度成正比的线性时间
18
19  template<class ForwardIterator1, class ForwardIterator2>
20  ForwardIterator1 search(ForwardIterator1 first1, ForwardIterator1 last1,
21                         ForwardIterator2 first2, ForwardIterator2 last2);
22  //检查[first2, last2]是否是在[first1, last1]范围内(是不是后者的一个子范围)
23  //如果是，就返回第一个匹配开始处的[first1, last1]中的一个迭代器
24  //如果没有找到匹配，就返回 last1
25  //时间复杂度：所需时间是[first, last]长度的 2 次方

26  template<class ForwardIterator, class T>
27  bool binary_search(ForwardIterator first, ForwardIterator last, const T& target);
28  //前条件：范围[first, last]用<按升序排序
29  //使用二叉搜索算法判断 target 是否在范围[first, last)中
30  //时间复杂度：对于随机访问迭代器，是 O(log N)
31  //对于非随机访问迭代器，则是线性运行时间 O(N)，N 是指范围[first, last]的长度
```

注意，若使用 binary_search 函数，可保证在实现中会采用二叉搜索算法，具体可参见第 14 章的讨论。二叉搜索算法的重要性在于，它能保证非常快的运行时间 $O(\log N)$。如果还没有学习第 14 章，或者还没有听说过二叉搜索，请将它想象成一种高效搜索算法。另外，它要求元素事先排好序。要理解本章的内容，只需知道二叉搜索的这两个特点。

√ 自测题

19. 在图 18.17 中，将标识符 vector 全文替换成 list。编译并运行程序。

20. 假定 v 是 vector<int>类的对象。使用 search 泛型函数(参见图 18.18)写一些代码，判断 v 是否包含连续的两个数字 42 和 43。不需要给出完整程序，但必须给出所有必要的 include 指令和 using 指令。**提示：**再用一个 vector 也许能简化编程。

会修改容器的算法

图 18.19 描述了 STL 中的一些泛型函数，它们能以某种形式更改容器内容。

记住，在容器中增删元素可能影响其他迭代器。一次增删操作后，并不保证迭代器还定位到以前那个元素，除非容器模板类做出了这样的保证。在前面讲过的模板类中，list 和 slist 能保证它们的迭代器不会因为添加或删除操作而移动，除非迭代器定位的就是要删除的元素。模板类 vector 和 deque 则无此保证。图 18.19 的某些函数模板保证了一些特定迭代器的值。无论容器是什么，都可信赖这些保证。

图 18.19 能修改容器的泛型函数

迭代器类型参数的名称指出函数支持的迭代器的种类。记住，这些只是最小的迭代器要求。例如，正向迭代器、双向迭代器和随机访问迭代器其实都是 ForwardIterator

```
1  template <class T>
2  void swap(T& variable1, T& variable2);
3  // 交换 variable1 和 variable2 的值
```

```
4  template<class ForwardIterator, class ForwardIterator2>
5  ForwardIterator2 copy(ForwardIterator1 first1, ForwardIterator1 last1,
6                        ForwardIterator2 first2, ForwardIterator2 last2);
7  //前条件：[first1, last1]和[first2, last2]这两个范围的长度一样
8  //动作：将位置[first1, last1]的元素
9  //复制到位置[first2, last2]
10 //返回 last2
11 //时间复杂度：与范围[first, last]的长度成正比的线性时间
12 template<class ForwardIterator, class T>
13 ForwardIterator remove(ForwardIterator first, ForwardIterator last,
14                        const T& target);
15 //从范围[first, last]中删除等于 target 的元素
16 //容器长度不变
17 //等于 target 的那些被删除的值被移到范围[first, last]的末尾
18 //这个范围有一个迭代器 i，使所有有不等于 target 的值都在范围[first, i]中。返回 i。
19 //时间复杂度：与范围[first, last]的长度成正比的线性时间

20 template<class BidirectionalIterator>
21 void reverse(BidirectionalIterator first, BidirectionalIterator last)
22 //逆转范围[first, last]中元素的顺序
23 //时间复杂度：与范围[first, last]的长度成正比的线性时间

24 template<class RandomAccessIterator>
25 void random_shuffle(RandomAccessIterator first, RandomAccessIterator last);
26 //使用一个伪随机数生成器对范围[first, last]中的元素进行随机排序
27 //时间复杂度：与范围[first, last]的长度成正比的线性时间
```

自测题

21. 可以为 list 容器使用 random_shuffle 模板函数吗？

22. 可以为 vector 容器使用 copy 模板函数吗？(copy 要求正向迭代器，vector 提供了随机访问迭代器)

set 算法

图 18.20 展示了如何运用由 STL 定义的泛型 set 所支持的各种操作。注意，这些泛型算法假定容器中的元素已排好序。容器 set，map，multiset 和 multimap 都以排序顺序存储它们的元素。因此，图 18.20 的所有函数都适合这 4 个模板类容器。其他容器，比如 vector，则不按照排序顺序存储元素。这些函数应用于这些容器。之所以要求元素先排序，是因为算法能更高效地执行。

图 18.20 set 操作

这些操作适合 set，map，multiset，multimap(以及其他容器)，但并非适合所有容器。例如，不适合 vector，list 或 deque，除非其中的内容已经排好序。为了正常使用，这些容器中的元素必须以排序顺序存储。这些操作全都适用于正向迭代器，所以延伸适用于双向和随机访问迭代器(某些时候，甚至适合其他种类的迭代器，只是这里没有具体讨论)。

```
1  template<class ForwardIterator1, class ForwardIterator2>
2  bool includes(ForwardIterator1 first1, ForwardIterator1 last1,
3                ForwardIterator2 first2, ForwardIterator2 last2)
4  //如果范围[first2, last2]中的每个元素还出现在范围[first1, last1]中，就返回 true
5  //否则返回 false
6  //时间复杂度：与范围[first1, last1]和[first2, last2]两者长度之和成正比的线性时间

7  template<class ForwardIterator1, class ForwardIterator2, class ForwardIterator3>
8  void set_union(ForwardIterator1 first1, ForwardIterator1 last1,
9                 ForwardIterator2 first2, ForwardIterator2 lastt2
10                ForwardIterator3 result);
11 //取两个范围[first1, last1]和[first2, last2]的并集，并进行排序
12 //合并后的范围从 result 处开始排序
13 //时间复杂度：与范围[first1, last1]和[first2, last2]两者长度之和成正比的线性时间
```

```
14 template<class ForwardIterator1, class ForwardIterator2, class ForwardIterator3>
15 void set_intersection(ForwardIterator1 first1, ForwardIterator1 last1,
16                       ForwardIterator2 first2, ForwardIterator2 lastt2
17                                            ForwardIterator3 result);
18 //取两个范围[first1, last1]和[first2, last2]的交集，并进行排序
19 //取交集后的范围从 result 处开始排序
20 //时间复杂度：与范围[first1, last1]和[first2, last2]两者长度之和成正比的线性时间
21 template<class ForwardIterator1, class ForwardIterator2, class ForwardIterator3>
22 void set_difference(ForwardIterator1 first1, ForwardIterator1 last1,
23                     ForwardIterator2 first2, ForwardIterator2 lastt2
24                     ForwardIterator3 result);
25 //取两个范围[first1, last1]和[first2, last2]的补集，并进行排序
26 //取补集之后的范围从 result 处开始排序
27 //时间复杂度：与[first1, last1]和[first2, last2]两者长度之和成正比的线性时间
```

√ 自测题

23. 数学课上学到的 set 不要求元素事先排好序，而且它有一个 union(联接)操作符。那么，为什么 set_union 模板函数要求容器以排好序的方式来存储元素呢？

排序算法

图 18.21 展示了两个模板函数的声明和文档，一个函数对一个范围中的元素排序，另一个则合并两个排好序的元素范围。注意，排序函数 sort 保证运行时间为 $O(\log N)$。虽然已超出本章范围，但确实能证明不能写比 $O(\log N)$还要快的、基于比较的排序算法。所以，这就保证了排序算法具有最快的速度(基于相同常量乘数)。

图 18.21　一些泛型排序算法

```
1  template<class RandomAccessIterator>
2  void sort(RandomAccessIterator first, RandomAccessIterator last);
3  //对范围[first, last]中的元素按升序进行排序
4  //时间复杂度：O(log N). 这里的 N 指的是[first, last]的长度

5  template<class ForwardIterator1, class ForwardIterator2, class ForwardIterator3>
6  void merge(ForwardIterator1 first1, ForwardIterator1 last1,
7             ForwardIterator2 first2, ForwardIterator2 last2
8             ForwardIterator3 result);
9  //前条件：对范围[first1, last1]和[first2, last2]排序
10 //动作：将两个范围合并成排好序的范围[result, last3]，其中：
11 //last3 = result + (last1 - first1) + (last2 - first2)
12 //时间复杂度：与[first1, last1]和[first2, last2]两个范围的长度之和成正比的线性时间
```

排序使用<操作符，所以<操作符必须定义。另一些版本(这里未给出)允许指定排序关系。“已排序”意味着已按升序排列

18.4　不断进化的 C++

C++在不断进化。ISO 专门有一个委员会审核 C++的提议修订。每隔几年就有新标准发布。本节简要介绍了 C++11 的一些新功能。注意这只是计算机编程和计算机科学更高级主题的一个入门指引。要深入学习，请参考更高级的参考书或网上的 ISO C++标准(https://isocpp.org/)。

std::array

标准容器 array 包含在<array>库中，允许使用向量风格的表示法随机访问固定大小的元素序列。容器允许像向量那样安全访问数组元素，但具有普通数组一样的性能和最小的存储要求。

图 18.22 展示了如何创建包含 6 个整数的数组，同时初始化前三个元素。剩余三个元素自动初始化为零，所以不会和普通数组一样有未知的、未初始化的值。

图 18.22　std::array

```
1 #include <iostream>
2 #include <array>
3
4 using std::cout;
5 using std::endl;
6 using std::array;
7
8 int main()
9 {
10    // 为数组分配空间来存储 6 个整数。
11    // 前三个主动设为 10, 20 和 30。
12    // 其他默认设为 0。
13    array<int,6> a = {10, 20, 30};
14
15    cout << "The size of the array: " << a.size() << endl;
16    cout << "The element at index 1: " << a[1] << endl;
17    cout << "Setting a[4] to 100" << endl;
18    a[4] = 100;
19    cout << "Outputting all elements of the array: " << endl;
20    for (int element : a)
21       cout << " " << element << endl;
22 }
```

示范对话

```
The size of the array: 6
The element at index 1: 20
Setting a[4] to 100.
Array contains:
10
20
30
0
100
0
```

可用 size()函数获取数组大小，这一点似向量但不似标准数组。还可用传统[]读取和设置数组内容。读取范围外的值返回 0，设置范围外的值则无效果。注意数组索引 3 和 5 设为默认值 0。适合向量的算法也可在这里使用。例如，如包含<algorithm>，就可对迭代器 begin()和 end()之间的元素进行排序。

```
std::sort(a.begin(), a.end());
cout << "After sort, array contains: " << endl;
for (int element : a)
   cout << element << endl;
```

输出如下所示：

```
After sort, array contains:
00
```

```
10
20
30
100
```

正则表达式

视频讲解：*Regular expression demonstration*

因篇幅所限，无法全面讨论正则表达式，这里只是简单总结了正则表达式并给出了几个 C++的例子。写作本书时，一些编译器还不支持 C++11 正则表达式库，所以检查你的编译器，看是否支持`<regex>`库。提醒熟悉正则表达式的读者，新 C++11 标准支持 Javascript 和 POSIX 格式。正式地说，正则表达式提供了描述正则语言的一种方式。考虑到本书的目的，我们认为正则表达式是描述模式的一种方式，可用于匹配文本序列。例如，可用正则表达式检查文本字符串是否包含 MM-DD-YYYY 格式的日期。没有正则表达式，就必须自己写代码来处理文本。模式很复杂，编码就很难。图 18.23 总结了基本正则表达式。

图 18.23　基本正则表达式

<table>
<tr><th>成员函数(c 是容器对象)</th><th>含义</th></tr>
<tr><td>字母或数字</td><td>相同的字母或数字。例如，正则表达式 a 匹配文本 a，正则表达式 abc123 匹配文本 abc123</td></tr>
<tr><td>.</td><td>匹配任意单字符</td></tr>
<tr><td>|</td><td>并集或逻辑 OR</td></tr>
<tr><td>R?</td><td>正则表达式 R 出现 0 次或 1 次</td></tr>
<tr><td>R+</td><td>正则表达式 R 连续重复 1 次或更多次</td></tr>
<tr><td>R*</td><td>正则表达式 R 连续重复 0 次或更多次</td></tr>
<tr><td>R{n}</td><td>正则表达式 R 连续重复 n 次</td></tr>
<tr><td>R{n,m}</td><td>正则表达式 R 连续重复 n 到 m 次</td></tr>
<tr><td>^</td><td>文本开始</td></tr>
<tr><td>$</td><td>文本结束</td></tr>
<tr><td>{元素列表}</td><td>匹配任何元素。例如，[abcd]匹配 a，b，c 或 d</td></tr>
<tr><td>{元素 1-元素 N}</td><td>匹配范围中的任何元素。例如，[a-za-Z]匹配任意大写或小写字母</td></tr>
<tr><td>()</td><td>优先级和表达式分组</td></tr>
</table>

一些简单的正则表达式例子

描述	正则表达式
三个 a 后跟三个 b	aaabbb 或 a{3}b{3}
零个或多个 a	a*
一个或多个 a，后跟零个或多个 b	a+b*
标识符规则；也就是说，字母或下画线后跟任意序列的字母、数字或下画线	[a-zA-Z_]+[a-zA-Z0-9_]*

C++11 regex 库包含许多有用的字符类，下面列出了一部分

正则表达式	含义
\d	单数字
\D	非数字
\s	空白字符(制表符、换行符、空格)
\w	单词字符(字母、数字或下画线)

可利用它们简化模式定义。例如，匹配“连续两个单词”这一模式可使用正则表达式 \w+\s\w+。它匹配的是“一个或多个单词字符，后跟空白字符，再后跟一个或多个单词字符”。

C++11 为了匹配正则表达式需包含<regex>库。regex 类是 std 命名空间的一部分，构造函数获取一个模式作为输入。regex 类的 regex_match 函数对模式和字符串进行全字匹配，regex_search 在字符串中搜索模式，regex_replace 在字符串中替换和模式匹配的内容。

图 18.24 展示了如何用 regex_match 判断 text1 或 text2 是否匹配“用空白分隔的两个单词”模式。注意由于需要在模式中包含字面值\，所以 C++11 新增的“原始字符串字面值”功能在这里非常好用(参见 2.2 节。要求字符串以 R 开头，字符串内容放到一对圆括号中)。不利用该功能，就需要写两个\\来转义单个\，第一个\是转义符。

图 18.24　正则表达式匹配

```
 1 #include <iostream>
 2 #include <regex>
 3 #include <string>
 4
 5 using std::cout;
 6 using std::getline;
 7 using std::cin;
 8 using std::endl;
 9 using std::string;
10 using std::regex;
11
12 int main()
13 {
14    // 格式为 xxx-xxx-xxxx 的电话号码
15    // R 代表原始字符串字面值，
16    // \字符不需要转义
17    string phonePattern = R"(\d{3}-\d{3}-\d{4})";
18    // 连续两个单词，中间用空白字符分隔
19    string twoWordPattern = R"(\w+\s\w+)";
20    regex regPhone(phonePattern);
21    regex regTwoWord(twoWordPattern);
22
23    string s;
24    cout << "Enter a string to test the phone pattern." << endl;
25    getline(cin, s);
26    if (regex_match(s, regPhone))
27       cout << s << " matches " << phonePattern << endl;
28    else
29       cout << s << " doesn't match " << phonePattern << endl;
30
31    cout << endl;
32    cout << "Enter a string to test the two word pattern." << endl;
33    getline(cin, s);
34    if (regex_match(s, regTwoWord))
35       cout << s << " matches " << twoWordPattern << endl;
36    else
37       cout << s << " doesn't match " << twoWordPattern << endl;
38 }
```

示范对话 1

```
Enter a string to test the phone pattern.
907-867-5309
907-867-5309 matches \d{3}-\d{3}-\d{4}
Enter a string to test the two word pattern.
word up
word up matches \w+\s\w+
```

示范对话 2

```
Enter a string to test the phone pattern.
867-5309
867-5309 doesn't match \d{3}-\d{3}-\d{4}
Enter a string to test the two word pattern.
oneword
oneword doesn't match \w+\s\w+
```

下面是匹配电话号码模式的另一个例子。要求用正则表达式匹配以下任意模式：

- (999) 999-9999
- 999-999-9999
- 999 999 9999

第一组三位数可用`\d`匹配数字，用`{3}`匹配三位数。

```
\d{3}
```

为匹配可选的左右圆括号，必须在圆括号前使用转义符，否则圆括号会被解释成优先级分组。在`\(`后添加`?`，匹配零个或一个左圆括号，在`\)`后添加`?`，匹配零个或一个右圆括号。所以，目前用于匹配前三位数有或没有圆括号的正则表达式是：

```
\(?\d{3}\)?
```

然后用短划线或空白字符分隔第一组和第二组数字。短划线或空白字符用正则表达式`(-|\s)`来匹配，整个表达式变成：

```
\(?\d{3}\)?(-|\s)
```

接着重复三位数：

```
\(?\d{3}\)?(-|\s)\d{3}
```

最后是短划线或空白字符加 4 位数：

```
\(?\d{3}\)?(-|\s)\d{3}(-|\s)\d{4}
```

以下代码输出"Phone number found"，因为`regex_search`在目标字符串的某个地方发现了和正则表达式匹配的内容。

```
string text = "Call me at (907) 867-5309";
string pattern = R"(\(?\d{3}\)?(-|\s)\d{3}(-|\s)\d{4})";
regex reg(pattern);

if (regex_search(text, reg))
   cout << "Phone number found" << endl;
```

最后，可用正则表达式迭代器找出和正则表达式匹配的每个地方。用`sregex_iterator`类遍历目标字符串中和正则表达式匹配的每个地方。C 风格字符串使用`regex_iterator`类。下例将显示字符串中的所有电话号码(共两处)。迭代器构造函数的参数包括正则表达式和对字符串首尾的引用。注意，`end_iterator`默认初始化为一个结束状态，可将此状态和`cur_iterator`进行对比。

```
string text =   "Call me at my desk phone (907) 867-5309 " +
                "or my cell phone 907-350-3491.";
string pattern = R"(\(?\d{3}\)?(-|\s)\d{3}(-|\s)\d{4})";
regex reg(pattern);

sregex_iterator cur_iterator(text.begin(), text.end(), reg);
sregex_iterator end_iterator;
while (cur_iterator != end_iterator)
{
    cout << cur_iterator->str() << endl;
    cur_iterator++;
}
```

示范对话

```
(907) 867-5309
907-350-3491
```

线程

视频讲解：*Threading demonstration*

线程是独立计算过程。C++支持编写多线程应用程序。可将线程视为并行的计算，多线程同时运行。处理器数量充足，这真的能行。但许多计算环境限制了并行运行能力。相反，计算机在不同线程之间切换资源，每个线程轮流运行一小段时间，给用户留下“并行运行”的错觉。

你其实随时都在体验多线程。现代操作系统允许同时运行多个程序。例如，无须等待查毒软件完成，随时都可以做其他事情，比如看电子邮件。操作系统利用线程来实现该功能。取决于计算机和操作系统，当前可能有、也可能没有并行的工作。更有可能的是，两个计算线程共享计算机资源并轮流运行。看电子邮件时，你可能注意不到因为要和查毒软件共享资源而造成的响应变慢。E-mail 软件确实会变慢，但由于计算机速度远超人的感知，所以几乎注意不到这个改变。

如需额外的速度并希望尽量并行计算，或者想在一个工作阻塞(比如等待用户输入)时继续处理，线程就相当有用。GPU 编程尤其适合成千上万个线程同时运行。支持 GPU 编程的服务器/工作站可媲美以前的超算。

和往常一样，这里只是用一些例子简单介绍线程。第一个例子(图 18.25)展示了如何用不同的线程运行一个函数。

图 18.25　多线程 Hello World

```
 1 #include <iostream>
 2 #include <thread>
 3
 4 using std::cout;
 5 using std::endl;
 6 using std::thread;
 7
 8 void func(int a)
 9 {
10     cout << "Hello World: " << a << " "
11     << std::this_thread::get_id() << endl;
12 }
```

```
13
14 int main()
15 {
16    thread t1(func, 10);
17    thread t2(func, 20);
18    t1.join();
19    t2.join();
20 }
```

示范对话

```
Hello World: 10 1399628350477824
Hello World: 20 1399628350583818
```

不同的线程 ID

编译该程序需链接到一个线程库。例如，在 Linux 环境中，g++编译器典型的命令行参数是：

```
g++ program.cpp -std=c++11 -pthread
```

程序启动两个线程。每个线程都运行函数 `func`。每个线程还自动获得唯一的 ID，可通过 `get_id()` 访问。运行程序将启动两个线程并分别输出“Hello World”。线程启动后我们无法控制它在什么时候运行，完全由操作系统决定！反复运行程序可体会到这一点。最终会看到一个线程的输出覆盖另一个的情况。这是由于一个线程正在输出时，可能发生上下文切换并运行第二个线程，从而覆盖第一个线程的输出。

`join()` 函数的作用是让 `main` 函数等待每个线程结束再继续。这是同步多个线程必须采取的措施。

要避免线程的输出相互覆盖，可添加一个 mutex(代表互斥，即 mutual exclusion)。它锁定线程，一次只允许一个线程进入一个代码区域。为防止死锁或其他类型的错误，这是相当重要的一个措施(学习操作系统时会更有体会)。图 18.26 修改了程序，强迫其他线程等待，确保任何时候只有一个线程运行 `func` 中的代码。

图 18.26　线程和 mutex

```
 1 #include <iostream>
 2 #include <thread>
 3 #include <mutex>
 4
 5 using std::cout;
 6 using std::endl;
 7 using std::thread;
 8 using std::mutex;
 9
10 mutex globalLock;
11
12 void func(int a)
13 {
14    globalLock.lock();
15    cout  << "Hello World: " << a << " "
16          << std::this_thread::get_id() << endl;
17    globalLock.unlock();
18 }
19 int main()
20 {
21    thread t1(func, 10);
22    thread t2(func, 20);
23    t1.join();
24    t2.join();
25 }
```

经常需要两个以上的线程。这时可以使用线程数组。以下代码生成 10 线程的一个数组：

```
thread tArr[10];
for (int i =0; i < 10; i++)
    tArr[i] = thread(func, i);
for (int i =0; i < 10; i++)
    tArr[i].join();
```

注意，无法预测哪个线程先运行：

```
Hello World: 0 140198342674176
Hello World: 3 140198311204608
Hello World: 2 140198321694464
Hello World: 4 140198300714752
Hello World: 1 140198332184320
Hello World: 5 140198290224896
Hello World: 6 140198279735040
Hello World: 7 140198269245184
Hello World: 8 140198258755328
Hello World: 9 140198248265472
```

可用线程运行一个类。图 18.27 给出了一个模板。本例将类命名为 Runnable，但实际任何名字都可以

图 18.27　在线程中运行类

```
 1 #include <iostream>
 2 #include <thread>
 3
 4 using std::cout;
 5 using std::endl;
 6 using std::thread;
 7 class Runnable
 8 {
 9 public:
10     Runnable();
11     Runnable(int n);
12     void operator()(); // 注意两个()
13 private:
14     int num;
15 };
16
17 Runnable::Runnable() : num(0)
18 {
19 }
20 Runnable::Runnable(int n) : num(n)
21 {
22 }
23 void Runnable::operator()()
24 {
25     cout << "Hello world, I am number " << num << endl;
26 }
27
28 int main()
29 {
30     Runnable r1(10);
31     Runnable r2(20);
32
33     thread t1(r1);
34     thread t2(r2);
35
36     t1.join();
37     t2.join();
38 }
```

线程启动时，Runnable 类执行 operator()() 方法中的代码。要传给线程的所有数据一般通过构造函数传递。

图 18.28 展示了最后一个例子。该程序创建三个线程，并行搜索数组的一部分来查找最小值。每个线程找到的最小值存储到 results 数组中的专有位置。main 函数必须遍历 results 查找最终的最小值。

创建线程时，传递的参数包括要搜索的数组、results 数组、ID 以及每个线程所搜索部分的边界。然后，每个线程搜索自己的那部分数组，找出最小值，并根据 ID 判断要将结果存储到 results 数组的哪个位置。

图 18.28　用多个线程搜索数组

```
 1 #include <iostream>
 2 #include <thread>
 3
 4 using std::cout;
 5 using std::endl;
 6 using std::thread;
 7
 8 class Runnable
 9 {
10 public:
11     Runnable();
12     Runnable(int *target, int *results, int num, int start, int end);
13     void operator()();
14 private:
15     int *target, *results;
16     int num, start, end;
17 };
18
19 Runnable::Runnable()
20 {
21     target=nullptr;
22     results=nullptr;
23     num=0;
24     start=0;
25     end=0;
26 }
27
28 Runnable::Runnable(int *target, int *results, int num, int start,
29                    int end)
30 {
31     this->target= target;
32     this->results = results;
33     this->num = num;
34     this->start = start;
35     this->end = end;
36 }
37
38
39 void Runnable::operator()()
40 {
41     int min = target[start];
42     for (int i=start+1; i<=end; i++)
43     {
44         if (target[i]<min)
45             min = target[i];
46     }
47     results[num] = min;
48 }
49
50 int main()
51 {
```

```
52     thread tarr[3];
53     int target[] = {31, 66, 41, 8, 92, 47, 22, 87, 45, 92, 4, 14};
54     int results[] = {999, 999, 999, 999};
55     for (int i = 0; i < 3; i++)
56     {
57        Runnable r(target, results, i, i*4, i*4+3);
58        tarr[i] = thread(r);
59     }
60     for (int i =0; i < 3; i++)
61        tarr[i].join();
62     for (int i =0; i < 3; i++)
63        cout << results[i] << endl;
64     int min = results[0];
65     if (min > results[1])
66        min = results[1];
67     if (min > results[2])
68        min = results[2];
69     cout << "The minimum from threaded min-search is " << min << endl;
70 }
```

示范对话

```
8
22
4
The minimum from threaded min-search is 4
```

智能指针

视频讲解：*Smart Pointers Demonstration*

第 9 章和第 13 章说了指针的好处，也指出在内存管理不恰当时会发生的问题。虚悬指针或内存泄漏会造成难以发现的错误。C++11 用新类 shared_ptr 简化了内存管理以及对象在内存中的共享。shared_ptr 是一个模板，是从自由存储分配的对象的包装器。包装器通过引用计数来跟踪其他还有多少个指针在引用对象。计数器从零开始。每次有一个新变量引用对象就递增 1。类似地，每次有一个变量不再引用对象就递减 1。换言之，变量被删除或重新赋值，计数器就递减。计数器归零，对象就可安全删除，分配的内存归还给自由存储。全部都自动进行，免于程序员写自己的内存管理代码。

例如，以下代码实现 Node 类的一个简单链表。类中只存储了一个整数。代码用指针链接类这一“旧”格式，不显式释放 listTest 函数中分配的内存。这意味着执行返回 main 函数会发生内存泄漏。程序不立即退出，就会造成内存问题。

图 18.29　存在内存泄漏的节点链表

```
 1 // 用传统指针实现的简单 Node 类的链表，
 2 // 注意该版本在执行返回 main 后会发生内存泄漏
 3
 4 #include <iostream>
 5 using std::cout;
 6 using std::endl;
 7
 8 // 一个简单 Node 类。全能型的类
 9 // 会多几个函数。
10 class Node
11 {
12 private:
13     int num;
14     Node *next;
```

```
15 public:
16     Node();
17     Node();
18     Node(int num, Node *nextPtr);
19     int getNum();
20     Node* getNext();
21     void setNext(Node *nextPtr);
22 };
23
24 Node::Node() : num(0), next(nullptr)
25 { }
26
27 Node::Node(int numVal, Node *nextPtr) : num(numVal), next(nextPtr)
28 { }
29
30 Node::~Node()
31 {
32     cout << "Deleting " << num << endl;
33 }
34 int Node::getNum()
35 {
36     return num;
37 }
38 Node* Node::getNext()
39 {
40     return next;
41 }
42 void Node::setNext(Node *nextPtr)
43 {
44     next = nextPtr;
45 }
46
47 void listTest()
48 {
49     // 创建链表 10->20->30
50     Node *root = new Node(10, nullptr);
51     root->setNext(new Node(20, nullptr));
52     root->getNext()->setNext(new Node(30, nullptr));
53
54     // 输出列表
55     Node *temp;
56     temp = root;
57     while (temp != nullptr)
58     {
59         cout << temp->getNum() << endl;
60         temp = temp->getNext();
61     }
62 }
63
64 int main()
65 {
66     listTest();
67 }
```

示范对话

```
10
20
30
```

Node 类虽有析构器却从未调用。这是由于根本没有删除每个节点。listTest 中分配的内存一直未释放，造成 main 中发生内存泄漏。虽然本例没什么问题，因为程序马上就退出了(此时必然回收内存)。但如果调用了 listTest 后还需更多处理，就会遇到内存

问题。

下面用 shared_ptr 类改写程序。必须包含<memory>库。将所有 Node 指针都替换成 shared_ptr<Node>，如图 18.30 所示。

图 18.30　使用智能指针的节点链表

```
 1 // 用智能指针实现的简单 Node 类的链表，
 2 // 不会出现内存泄漏，因为 shared_ptr 类
 3 // 自动处理引用计数和内存释放。
 4 #include <iostream>
 5 #include <memory>
 6 using std::cout;
 7 using std::endl;
 8 using std::shared_ptr;
 9
10 // 修改了类来使用 Node 的 shared_ptr。
11 class Node
12 {
13 private:
14     int num;
15     shared_ptr<Node> next;
16 public:
17     Node();
18     ~Node();
19     Node(int num, shared_ptr<Node> nextPtr);
20     int getNum();
21     shared_ptr<Node> getNext();
22     void setNext(shared_ptr<Node> nextPtr);
23 };
24 Node::Node() : num(0), next(nullptr)
25 { }
26
27 Node::~Node()
28 {
29     cout << "Deleting " << num << endl;
30 }
31
32 Node::Node(int numVal, shared_ptr<Node> nextPtr) : num(numVal), next(nextPtr)
33 { }
34
35 int Node::getNum()
36 {
37     return num;
38 }
39
40 shared_ptr<Node> Node::getNext()
41 {
42     return next;
43 }
44
45 void Node::setNext(shared_ptr<Node> nextPtr)
46 {
47     next = nextPtr;
48 }
49
50 void listTest()
51 {
52     shared_ptr<Node> root(new Node(10, nullptr));
53     shared_ptr<Node> next1(new Node(20, nullptr));
54     shared_ptr<Node> next2;
55     // 声明好一个 shared_ptr 后，
56     // 可用 reset 函数设置它。
```

```
57     next2.reset(new Node(30, nullptr));
58     // 链接所有节点
59     root->setNext(next1);
60     next1->setNext(next2);
61
62     // 输出列表
63     shared_ptr<Node> temp;
64     temp = root;
65     while (temp != nullptr)
66     {
67         cout << temp->getNum() << endl;
68         temp = temp->getNext();
69     }
70 }
71
72 int main()
73 {
74     listTest();
75     cout << "Exiting program." << endl;
76 }
```

示范对话

```
10
20
30
Deleting 10
Deleting 20
Deleting 30
Exiting program.
```

一旦变量离开 listTest 函数的作用域，shared_ptr 类就自动回收链表。这在 listTest 调用退出后才发生，证据就是程序退出前由 Node 析构函数输出的消息。

再看看假如有一个全局变量引用链表的第二项会发生什么。此时 shared_ptr 类不会在 listTest 函数退出时删除剩余项。这是由于节点只有在零引用时才会删除。注意使用全局变量并不是一个好的编程实践。这里只是为了演示引用计数概念。新的全局变量如下：

```
shared_ptr<Node> global_reference;
```

listTest 函数的代码进行以下修改：

```
void listTest()
{
    shared_ptr<Node> root(new Node(10, nullptr));
    shared_ptr<Node> next1(new Node(20, nullptr));
    shared_ptr<Node> next2;
    // 声明好一个 shared_ptr 后，
    // 可用 reset 函数设置它。
    next2.reset(new Node(30, nullptr));
    // 链接所有节点
    root->setNext(next1);
    next1->setNext(next2);
    // Output the list
    shared_ptr<Node> temp;
    temp = root;
    while (temp != nullptr)
    {
        cout << temp->getNum() << endl;
        temp = temp->getNext();
```

```
    }
    // The line below creates a reference to the second item
    // in the linked list
    global_reference = root->getNext();
}
```

示范对话

```
10
20
30
Deleting 10
Exiting program.
Deleting 20
```

最大区别就是当 listTest 函数退出时只有第一个节点才会删除，因其没有任何引用。剩余两个节点因为全局变量的关系仍然有引用。但是，当程序最终退出时，就连这些节点也会离开作用域，内存会被回收。

注意 shared_ptr 类并非万能。循环引用列表会出问题，因为引用计数永远到不了 0，内存会一直回收不了。C++11 解决该问题的方案是提供一个额外的 weak_ptr 类。只要 weak_ptr 是唯一的对象引用，该对象就会被销毁。只要至少一个链接由 weak_ptr 连接，整个循环列表最终都会被销毁。

C++11 还提供了 unique_ptr 类，不能把它赋给其他任何指针。旧版本 C++支持一个名为 auto_ptr 的类，但已在 C++中废弃。

小　结

- 迭代器是指针的泛化形式。迭代器在容器的某个范围中遍历元素。通常为迭代器定义了++，--和*(提领)等操作。
- 有迭代器的容器类定义了成员函数 end()和 begin()，它们返回迭代器值，便于你处理容器中的所有元素，如下所示：

```
for (p = c.begin(); p != c.end(); p++)
  处理 *p // *p 是当前数据项
```

- 迭代器的主要种类包括：
  正向迭代器：支持++
  双向迭代器：支持++和--
  随机访问迭代器：支持++，--和随机访问
- 对于常量迭代器 p，提领操作符*p 生成的是元素的只读版本。对于可变迭代器 p，则可以为*p 赋值。
- 双向容器有逆向迭代器，允许代码按相反顺序遍历容器中的元素。
- STL 中的主要容器模板类是 list(它有可变双向迭代器)，另外还有 vector 和 deque(两者都有可变随机访问迭代器)。
- stack 和 queue 是容器配接器类，意味着它们在其他容器类的顶部构建。stack 是后入/先出容器。queue 则是先入/先出容器。
- set，map，multiset 和 multimap 容器模板类以排好序的方式来存储元素，这是出于对搜索算法效率的考虑。set 是简单的元素集合。map 允许根据键来存储和检索元素。multiset 类允许存储重复的项。multimap 类则允许将多个数据项关联到一个键。
- STL 包含大量模板函数，它们实现了泛型算法，其最大运行时间有保障。
- C++11 新功能包括 std::array 类、正则表达式、多线程和智能指针。

自测题答案

1. v.begin()返回定位到 v 的第一个元素的迭代器。v.end()则返回哨兵值，标志 v 的所有元素的末尾。
2. *p 是应用于 p 的提领操作符。*p 是对位置 p 的元素的引用。
3.
```
vector<int>::iterator p;
for (p = v.begin(), p++; p != v.end(); p++)
    cout << *p << " ";
```
4. D C C
5. B C
6. 两种方式都可以。
7. 主要区别是 vector 容器有随机访问迭代器，list 只有双向迭代器。
8. 除 slist 之外的所有模板类。
9. vector 和 deque。
10. 都有可变迭代器。
11. stack 模板配接器类没有迭代器。
12. queue 模板配接器类没有迭代器。
13. 没有返回值；pop 是 void 函数。
14. mymap 会包含两项。一个是从 5 到"c++"的映射，另一个是从 4 到默认字符串(空字符串)的映射。
15. 是的，set 可包含任何类型的元素，虽然每个 set 对象只能包含一种类型的元素。模板类的类型参数就是存储的元素的类型。
16. 如果'A'在 s 中，s.find('A')返回一个迭代器，它定位到元素'A'；如果'A'不在 s 中，s.find('A')返回 s.end()。
17. 只要 $1 <= N$，那么 $aN + b <= (a + b)N$。
18. 这是数学问题，和 C++语言无关。所以，=的意思是等于，而不是赋值。
    首先，请注意 $\log_a N = (\log_a b)(\log_b N)$。
    求 a 的 $\log_a N$ 次方，结果是 N；求 a 的$(\log_a b)(\log_b N)$次方，结果也是 N。
    设 $c = (\log_a b)$，那么 $\log_a N = c\ (\log_b N)$。
19. 程序运行结果没什么不同。
20.
```
#include <iostream>
#include <vector>
#include <algorithm>
using std::cout;
using std::vector;
using std::search;
   ...
vector<int> target;
target.push_back(42);
target.push_back(43);
vector<int>::const_iterator result = search(v.begin(), v.end(),
                   target.begin(), target.end());
```

```
if (result != v.end())
    cout << "Found 42, 43.\n";
else
    cout << "42, 43 not there.\n";
```

21. 不可以，这要求有随机访问迭代器；list 模板类只有双向迭代器。
22. 是的，随机访问迭代器同时也是正向迭代器。
23. set_union 模板函数要求容器以排好序的方式存储其元素，确保函数模板能高效地实现。

编程练习

编程练习一般只需写很小的程序，运用本章提到的编程概念。

1. 写程序声明一个 deque 来存储 double 类型的值。读入 10 个 double 值，将它们存储到 deque 中。然后，调用泛型函数 sort 对 deque 中的数进行排序，显示结果。
2. 视频讲解：*Solution to Practice Program 18.2*

   写程序用 map 模板类统计用户输入的正数。在映射中，键是输入的数字，值是当时输入的键的个数。输入-1 表示输入结束。例如，如输入以下数字：

   ```
   5
   12
   3
   5
   5
   3
   21
   -1
   ```

   程序应显示以下输出(不一定是这个顺序)：

   ```
   The number 3 occurs 2 times.
   The number 5 occurs 3 times.
   The number 12 occurs 1 times.
   The number 21 occurs 1 times.
   ```

3. 给定设为任意文本的 string 变量，写程序用 char 类型的 stack 模板类反转字符串。
4. 假定一个列表包含学生 ID，后跟学生选修课程编号(以空格分隔)。列表未排序。例如，假定学生 1 选修了 CS100 和 CS200，学生 2 选修了 CS105 和 MATH210，那么列表可能是下面这样：

   ```
   1 CS100
   2 MATH210
   2 CS105
   1 CS200
   ```

   写程序从控制台读取这种格式的数据。ID 是-1 就终止输入。使用 map 模板类将整数(学生 ID)映射到容纳了学生选修的每门课的字符串向量。所有数据输入完毕后，遍历 map 并输出学生 ID 及其选修的所有课程。输出的是按学生 ID 排序的课程列表。

   如 C++版本低于 C++11，不要忘了定义向量映射时需在>>之间插入空格。

编程项目

编程项目要求综合运用多方面的知识来解决问题，程序一般比编程练习大，解题方式多样化。

1. 写程序允许用户输入任意数量的学生姓名及其分数。然后，程序应显示学生姓名及其分数(按分数升序排列)。使用 STL 的模板类 vector 和泛型 sort 函数。注意，需要为数据定义一个结构或者类类型，

以便在其中包含学生姓名和分数。另外，还要为这个结构或类重载<操作符。

2. 质数是大于 1 而且只能被它自身和 1 除尽的整数。对于三个整数 x，y 和 z，如果 $x = y * z$，那么整数 x 就可以被整数 y 整除。古希腊数学家埃拉托色尼(Erathosthenes)发明了一种算法，它能查找小于整数 N 的所有质数。这个算法称为素数筛(Sieve of Erathosthenes)。原理如下：首先建立一个包含整数 2～N 的列表。数字 2 是第一个质数，2 的倍数(也就是 4，6，8……，等等)不是质数。我们把它们从列表中剔除。然后，2 之后第一个未被剔除的数是第二个质数。这个数是 3。3 的倍数不是质数。从列表中剔除 3 的倍数。注意，6 在之前已经剔除，所以第一个剔除的是 9。同样地，12 已经在之前剔除，所以第二个剔除的是 15；依此类推。在 3 之后，第一个未被剔除的数是下一个(第三个)质数。这个算法将一直像这样推算，直到最终抵达数字 N。到此为止，所有没有被剔除的数都是质数。

   a. 写一个程序，利用上述算法来找出小于用户指定数 N 的所有质数。请使用包含了整数的一个 vector 容器。使用由 bool 值构成的一个数组，将元素事先全部初始化为 true，并用它来跟踪被剔除的整数。一旦某个整数从列表中剔除，与之对应的数组元素就变成 false。

   b. 测试 $N = 10$，30，100 和 300。

   c. 实际上，我们并不需要一直推算到 N。在 $N/2$ 的时候就可以停止了。请对此进行试验，并重新测试程序。$N/2$ 能够提升程序的效率，但却并不是我们可以使用的最小的数。请自行证明，为了计算 1～N 的所有质数，最小的推算上限是 N 的平方根。

   d. 修改(a)部分的代码，将 N 的平方根作为上限。

3. 假定有一个学生记录集。记录采用结构的形式：

```
struct StudentInfo
{
    string name;
    int grade;
};
```

记录是放在一个 vector<StudentInfo>中维护的。写程序提示输入数据，根据数据来构建由学生记录构成的 vector，按姓名对 vector 进行排序，计算最高、最低成绩以及班级平均成绩，并打印这些总结数据和附有成绩的班级名册(不要求打印具体是谁获得了最高分、最低分，只需打印最高分、最低分和平均分)。测试程序。

4. 继续编程项目 3，写函数将由 StudentInfo 记录构成的 vector 中的学生划分到两个 vector 中。一个 vector 包含及格学生的记录，另一个 vector 包含不及格学生的记录(60 分作为及格分数线)。可采用两种方式完成这个项目，要求分别给出运行时间统计。

   a. 可考虑继续使用一个 vector。可生成第二个 vector 包含及格学生，第三个 vector 包含不及格学生。但是，这样做肯定会在一定时间内存储重复的记录，所以最好不要这样做。可创建一个 vector 专门存储不及格学生，并写一个测试不及格学生的函数。然后，用 push_back 操作将不及格学生存储到 vector 中，从原始 vector 中删除(使用 erase 成员函数)不及格学生的记录。

   b. 思考这种方案的效率。可能要从 vector 的中部删除 n 个成员。在这种情况下，必须移动大量成员。从 vector 中部删除是 O(*N*)操作。给出这个程序的 big-O 运行时间估计。

   c. 如果使用一个 list<StudentInfo>，erase 和 insert 函数的运行时间是什么？思考在 list 中使用 erase 的效率会对程序的运行时间产生什么影响。使用 list(而不是 vector)重写这个程序。记住，list 既不支持索引访问，也不支持随机访问，而且它只有双向迭代器，没有随机访问迭代器。

5. 重做(或者第一次做)第 11 章的编程项目 9，这次要求使用 STL set 模板类，而不是你自己的集合类。使用泛型 set_intersection 函数计算 Q 和 D 的交集。

下例演示了如何使用 set_intersection 来求 A 和 B 的交集，并将结果保存到 C 中，其中所有集合都是字符串集合：

```
#include <iterator>
#include <set>
#include <string>
…
```

```
set<string> C;
// 注意下面这一行中，>和>之间的空格
insert_iterator<set<string> > cIterator(C, C.begin());
set_intersection(A.begin(), A.end(),
                 B.begin(), B.end(),
                 cIter);
// 集合 C 现在包含 A 和 B 的交集
```

6. 视频讲解：*Solution to Programming Project 18.6*

这个项目用向量创建书籍数据库。要跟踪每本书的作者、书名和出版日期。程序要提供主菜单，允许用户选择以下选项：(1) 添加一本书的作者、书名和日期；(2) 打印按作者姓名来排序的书籍列表；(3) 退出。

必须用一个类(book 类)容纳每本书的数据。这个类必须包含三个字符串字段，一个容纳作者姓名，一个容纳出版日期，另一个容纳书名。将整个书籍数据库保存到一个向量中，向量的每个元素都是一个 book 类对象。

为了对数据进行排序，请使用<algorithm>库提供的泛型 sort 函数。注意，这要求定义<操作符来比较 Book 类型的两个对象，使两本书的作者字段能够比较。

下面展示了示范性的输入/输出行为。你的 I/O 不一定和这里显示的相同，它只是让你了解程序的功能。

```
Select from the following choices:
1.  Add new book
2.  Print listing sorted by author
3.  Quit
1

Enter title:
More Than Human

Enter author:
Sturgeon, Theodore

Enter date:
1953

Select from the following choices:
1.  Add new book
2.  Print listing sorted by author
3.  Quit
1

Enter title:
Problem Solving with C++

Enter author:
Savitch, Walter

Enter date:
2015

Select from the following choices:
1.  Add new book
2.  Print listing sorted by author
3.  Quit
2

The books entered so far, sorted alphabetically by author are:
   Savitch, Walter, Problem Solving with C++, 2015
   Sturgeon, Theodore, More Than Human, 1953
Select from the following choices:
1.  Add new book
2.  Print listing sorted by author (case-sensitive)
3.  Quit
1
Enter title:
```

```
At Home in the Universe
Enter author:
Kauffman

Enter date:
1996

Select from the following choices:
1.  Add new book
2.  Print listing sorted by author (case-sensitive)
3.  Quit
2

The books entered so far, sorted alphabetically by artist are:
   Kauffman. At Home in the Universe, 1996
   Savitch, Walter. Problem Solving with C++, 2015
   Sturgeon, Theodore. More Than Human, 1953
```

7. 重做(或者第一次做)第 14 章的编程项目 8，这次为所有集合操作都使用 STL 集合类，并用 STL 链表类来存储和操纵每个单独的排列组合。创建集合来容纳列表时，记住在最后两个>之间插入一个空格，否则编译器可能会被搞糊涂。例如，set<list<int> >定义了其元素是链表的一个集合，链表中包含 int 类型的元素。而忘记插入空格的 set<list<int>>可能造成编译错误。(从 C++11 开始解决了这个问题。)

8. 你收集了一个电影评分文件，其中每部电影都被打了从 1(很差)到 5(完美)的一个分。文件第一行是一个数字，它指出文件中总共保存了多少个评分。在它下面，每个评分都由两行组成。第一行是电影名，第二行是 1~5 的评分。下面是一个示例评分文件，它包含 4 部不重复的电影和 7 个评分：

```
7
Harry Potter and the Order of the Phoenix
4
Harry Potter and the Order of the Phoenix
5
The Bourne Ultimatum
3
Harry Potter and the Order of the Phoenix
4
The Bourne Ultimatum
4
Wall-E
4
Glitter
1
```

写程序读取这种格式的文件，计算每部电影的平均分，输出平均分以及这部电影在文件中被打了多少次分。下面是基于上述示例数据的示例输出：

```
Glitter: 1 review, average of 1/5
Harry Potter and the Order of the Phoenix: 3 reviews, average of 4.3 / 5
The Bourne Ultimatum: 2 reviews, average of 3.5 / 5
Wall-E: 1 review, average of 4 / 5
```

用一个或多个映射来计算输出。要建立从“电影名称”字符串到“打分次数”整数和“总分”整数的映射。

9. 考虑一个姓名文本文件，每个姓名一行，姓名来自几个不同的来源。一个例子是：

```
Brooke Trout
Dinah Soars
Jed Dye
Brooke Trout
Jed Dye
Paige Turner
```

文件存在重复姓名。现在要生成一个邀请列表，但不想向同一个人发送多份邀请。写程序用 set 模板类删除重复姓名。从文件读取每个姓名，添加到 set，输出 set 中的所有姓名，以生成无重复的邀请列表。

10. 重做第 8 章的编程项目 16，这次用 Racer 类存储每个运动员的信息。类中存储了运动员的姓名、编

号、最终排名和 RFID 传感器记录的所有分段计时。可选择合适的结构存储这些信息。添加恰当的函数来获取或更改运动员信息，还要添加一个构造函数。

用 map 存储比赛数据。map 使用运动员的号码作为键，使用和号码对应的 Racer 对象作为值。使用 map 就不需要在多个重复的号码中搜索了，可直接根据号码访问分段计时和最终排名。

如版本低于 C++11，定义向量 map 时不要忘记在>>字符之间添加空格。

11. 写程序用单独的线程运行计数器。计数器从 1 开始，每秒递增 1，每 5 秒向控制台输出值。下面这行 C++11 代码使当前线程等待 1 秒，需要包含<chrono>：

```
std::this_thread::sleep_for(std::chrono::seconds(1));
```

计数器线程运行时，主线程允许输入一个数，如输入的数小于等于计数器的值，程序终止。

12. 写程序用正则表达式校验 MM/DD/YY 格式的日期。 YY 必须是两位数，但 MM 可以是 1~9 的单个数(例如，用 1 而不是 01 表示一月)，也可以是 01~12 的两位数。MM 中的数不能超过 12；例如，13 为无效。类似地，DD 可以是 1~9 的单个数，也可以是 01~31 的两位数。DD 中的数不能超过 31；例如，32 为无效。不必关心小于 31 天的月份。例如，2 月 31 号为无效日期，但本题可认为有效。

附录 A　C++关键字

在 C++中，以下关键字具有预定义的用途，不能挪作他用。具体地说，不可将其用于变量名和程序员自定义函数。除了下面列出的关键字，含双下画线(__)的标识符保留给 C++实现和标准库使用，不应在程序中使用。

alignas	default	if	reinterpret_cast	try
alignof	delete	inline	return	typedef
asm	do	int	short	typeid
auto	double	log	signed	typename
bool	dynamic_cast	long	sizeof	union
break	else	mutable	static	unsigned
case	enum	namespace	static_assert	using
catch	explicit	new	static_cast	virtual
char	export	noexcept	struct	void
class	extern	nullptr	switch	volatile
const	false	operator	template	wchar_t
const_cast	float	private	this	while
constexpr	for	protected	thread_local	
continue	friend	public	throw	
decltype	goto	register	true	

操作符和标点符号的以下替代表示法也是保留的，不能挪作他用。

and &&	and_eq &=	bitand &	bitor |	coml ~	not !
not_eq !=	or ||	or_eq |=	xor ^	xor_eq ^=	

附录 B　操作符的优先级

如下所示，同一个框内的所有操作符具有相同的优先级。较高的框中的操作符的优先级高于较低的框中的操作符。操作符具有相同优先级时，一元操作符和赋值操作符从右向左执行。例如，x = y = z 实际是 x = (y = z)。具有相同优先级的其他操作符则从左向右执行。例如，x + y + z 实际是(x + y) + z。

最高优先级（最先计算）

<table>
<tr><td>::　作用域解析操作符</td></tr>
<tr><td>.　圆点操作符
->　成员选择
[]　数组索引
()　函数调用
++　后递增操作符（放在变量之后）
--　后递减操作符（放在变量之后）</td></tr>
<tr><td>++　前递增操作符（放在变量之前）
--　前递减操作符（放在变量之前）
!　求反
-　一元减
+　一元加
*　提领
&　取址
new
delete
delete[]
sizeof</td></tr>
<tr><td>*　乘
/　除
%　求余（取模）</td></tr>
<tr><td>+　加
-　减</td></tr>
<tr><td><<　插入操作符（输出）
>>　提取操作符（输入）</td></tr>
<tr><td><　小于　<=　小于或等于
>　大于　>=　大于或等于</td></tr>
<tr><td>==　等于
!=　不等于</td></tr>
<tr><td>&&　逻辑 AND（逻辑与）</td></tr>
<tr><td>||　逻辑 OR（逻辑或）</td></tr>
<tr><td>=　赋值
+=　加后赋值　-=　减后赋值
*=　乘后赋值　/=　除后赋值　%=　取模后赋值</td></tr>
</table>

最低优先级（最后计算）

附录 C　ASCII 字符集

下表展示了可打印字符。编号为 32 的字符是空格。

32		56	8	80	P	104	h
33	!	57	9	81	Q	105	i
34	"	58	:	82	R	106	j
35	#	59	;	83	S	107	k
36	$	60	<	84	T	108	l
37	%	61	=	85	U	109	m
38	&	62	>	86	V	110	n
39	'	63	?	87	W	111	o
40	(	64	@	88	X	112	p
41	)	65	A	89	Y	113	q
42	*	66	B	90	Z	114	r
43	+	67	C	91	[	115	s
44	,	68	D	92	\	116	t
45	-	69	E	93	]	117	u
46	.	70	F	94	^	118	v
47	/	71	G	95	_	119	w
48	0	72	H	96	`	120	x
49	1	73	I	97	a	121	y
50	2	74	J	98	b	122	z
51	3	75	K	99	c	123	{
52	4	76	L	100	d	124	|
53	5	77	M	101	e	125	}
54	6	78	N	102	f	126	~
55	7	79	O	103	g		

附录 D　部分库函数

下面几个表根据函数用途(而不是所属的库)进行分类。函数声明给出了参数的数目和类型，还给出了返回值类型。大多数函数声明只给出参数类型，不给出参数名称(4.3 节对这种函数声明进行了解释)。

数学函数

函数声明	说明	头文件名称
int abs(***int***);	绝对值	cstdlib
long labs(***long***);	绝对值	cstdlib
double fabs(***double***);	绝对值	cmath
double sqrt(***double***);	平方根	cmath
double pow(***double***, ***double***);	返回第一个参数的第二个参数次幂	cmath
double exp(***double***);	返回指数，以 *e* 为底	cmath
double log(***double***);	自然对数(ln)	cmath
double log10(***double***);	以 10 为底的对数	cmath
double ceil(***double***);	返回大于或等于参数的最小整数	cmath
double floor(***double***);	返回小于或等于参数的最大整数	cmath

输入和输出成员函数

函数调用形式	说明	头文件
Stream_Var.open (*External_File_Name*);	把名为 *External_File_Name* 的文件连接到 *Stream_Var* 所命名的流。*External_File_Name* 是一个字符串值	fstream
Stream_Var.fail();	如果对流 *Stream_Var* 的前一个操作(比如打开)失败，就返回 true	fstream 或 iostream
Stream_Var.close();	将流 *Stream_Var* 与其连接到的文件断开	fstream
Stream_Var.bad();	如果流 *Stream_Var* 被破坏，就返回 true	fstream 或 iostream
Stream_Var.eof();	程序在读入与输入流 *Stream_Var* 连接的那个文件中的最后一个字符之后，如果还在尝试读取，就返回 true；否则返回 false	fstream 或 iostream
Stream_Var.get (*Char_Variable*);	从输入流 *Stream_Var* 读取一个字符，并将 *Char_Variable* 设置为等于这个字符。不忽略空白字符	fstream 或 iostream
Stream_Var.getline (*String_Var*, *Max_Characters* +1);	从流 *Stream_Var* 中读取一行输入，结果字符串放到 *String_Var* 中。如果这一行的长度超过了 *Max_Characters*，就只读取前 *Max_Characters* 个字符。*String_Var* 的声明长度应大于或等于 *Max_Characters* + 1	fstream 或 iostream

续表

函数调用形式	说明	头文件
Stream_Var.peek();	从输入流 *Stream_Var* 中读取一个字符，然后返回该字符。但读取的字符不从输入流中删除；下次读取的仍然是同一个字符	fstream 或 iostream
Stream_Var.put(*Char_Exp*);	把 *Char_Exp* 的值写到输出流 *Stream_Var* 中	fstream 或 iostream
Stream_Var.putback(*Char_Exp*);	把 *Char_Exp* 的值放入输入流，使那个值成为从流中读取的下一个输入值。连接到流的文件不会改变	fstream 或 iostream
Stream_Var.precision(*Int_Exp*);	为发送到输出流 *Stream_Var* 的浮点数指定小数点之后的位数	fstream 或 iostream
Stream_Var.width(*Int_Exp*);	为输出到流 *Stream_Var* 的下一个值设置域宽	fstream 或 iostream
Stream_Var.setf(*Flag*);	设置用于格式化到流 *Stream_Var* 的输出的标志。图 6.5 总结了这些标志	fstream 或 iostream
Stream_Var.unsetf(*Flag*);	取消设置用于格式化到流 *Stream_Var* 的输出的标志。图 6.5 总结了这些标志	fstream 或 iostream

字符函数

所有情况下，参数实际类型是 int，但针对大多数用途，都可将参数类型视为 char。如返回 int，必须执行显式或隐式类型转换才能获得一个 char。

函数声明	说明	头文件
bool isalnum(**char**);	如果参数满足 isalpha 或 isdigit，就返回 true；否则返回 false	cctype
bool isalpha(**char**);	如果参数是一个大写或小写字母，就返回 true(此外，它还可能为其他参数返回 true。这具体要依赖于实现)；否则返回 false	cctype
bool isdigit(**char**);	如果参数是一个数字，就返回 true；否则返回 false	cctype
bool ispunct(**char**);	如果参数是一个可打印字符，但它不满足 isalnum，也不是一个空白字符，就返回 true；否则返回 false。换言之，该函数在参数是一个标点符号的前提下返回 true。	cctype
bool isspace(**char**);	如果参数是一个空白字符(比如空格、制表符或换行符)，就返回 true；否则返回 false	cctype
bool iscntrl(**char**);	如果参数是一个控制字符，就返回 true；否则返回 false	cctype
bool islower(**char**);	如果参数是一个小写字母，就返回 true；否则返回 false	cctype
bool isupper(**char**);	如果参数是一个大写字母，就返回 true；否则返回 false	cctype
int tolower(**char**);	返回参数的小写形式。如果没有相应的小写形式，就返回其参数本身	cctype
int toupper(**char**);	返回参数的大写形式。如果没有相应的大写形式，就原封不动地返回参数	cctype

字符串函数

函数声明	说明	头文件
int atoi(***const char*** a[]);	将字符串转换为 int	cstdlib
int stoi(***const*** string)	将 STL 字符串对象转换为 int。C++11 和更高版本	string
long atol(***const char*** a[]);	将字符串转换为 long	cstdlib
long stol(***const*** string)	将 STL 字符串对象转换为 long。C++11 和更高版本	
double atof(***const char*** a[]);	将字符串转换为 *double*	cstdlib①
strcat(String_Variable, String_Expression);	把 *String_Expression* 的值追加到 *String_Variable* 中的字符串的末尾	cstring
strcmp(String_Exp1, String_Exp2)	如果两个字符串表达式的值不同，就返回 *true*；否则返回 *false*②	cstring
strcpy(String Variable, String_Expression);	将 *String Variable* 的值更改为 *String Expression* 的值	cstring
strlen(String_Expression);	返回 *String_Expression* 的长度	cstring
strncat(String_Variable, String_Expression, Limit);	与 *strcat* 一样，但最多追加 *Limit* 个字符	cstring
strncmp(String_Exp1, String Exp2, Limit);	与 *strcmp* 一样，但最多比较 *Limit* 个字符	cstring
strncpy(String Variable, String_Expression, Limit);	和 *strcpy* 一样，但最多复制 *Limit* 个字符	cstring
strstr(String_Expression, Pattern)	返回一个指针，它指向 *String_Expression* 中出现的第一个字符串 *Pattern*。如果没有找到 *Pattern*，就返回 *NULL* 指针	cstring
strchr(String_Expression, Character)	返回一个指针，它指向 *String_Expression* 中出现的第一个 *Character*。如果没有找到 *Character*，就返回 *NULL* 指针	cstring
strrchr(String_Expression, Character)	返回一个指针，它指向 *String_Expression* 中出现的最后一个 *Character*。如果没有找到 *Character*，就返回 *NULL* 指针	cstring

①有的实现将其放入 cmath。

②根据 *String_Exp1* 是小于、等于或大于 *String_Exp2*，分别返回一个小于零、等于零或大于零的值。依据词典顺序。

随机数生成器

函数声明	说明	头文件
int random(***int***);	调用 random(n)，会返回大于或等于 0，同时小于或等于 n-1 的一个伪随机数(并非所有实现都支持。如果不支持，还必须换用 rand)	cstdlib

续表

函数声明	说明	头文件
int rand();	调用 rand()，会返回大于或等于 0，同时小于或等于 RAND_MAX 的一个伪随机数。RAND_MAX 是一个预定义的整数常量，在 cstdlib 中定义。RAND_MAX 的值依赖于实现，但至少为 32767	cstdlib
void srand(***unsigned int***); ***void*** srandom(***unsigned int***); (***unsigned int*** 类型是只允许非负值的整数类型。可看成是限制为非负的 ***int*** 类型)	重新初始化随机数生成器。参数值是种子值。用同样的参数多次调用 srand，会导致 rand 或 random(不管使用哪一个)生成同样的伪随机数序列。调用 rand 或 random 之前，如果没有调用过 srand，生成的随机数序列就等同于以前用参数值 1 调用了 srand	cstdlib

三角函数

这些函数使用的是弧度，不是角度。

函数声明	说明	头文件
double acos(***double***);	反余弦	cmath
double asin(***double***);	反正弦	cmath
double atan(***double***);	反正切	cmath
double cos(***double***);	余弦	cmath
double cosh(***double***);	双曲余弦	cmath
double sin(***double***);	正弦	cmath
double sinh(***double***);	双曲正弦	cmath
double tan(***double***);	正切	cmath
double tanh(***double***);	双曲正切	cmath

附录 E　内联函数

如果一个成员函数定义相当短，可在类定义中直接给出它的函数定义。只需将成员函数声明替换为成员函数定义；但由于定义包含在类定义中，所以不需要包含类名和作用域解析操作符。例如，下面定义的 Pair 类包含两个构造函数和一个成员函数 getFirst 的内联函数定义：

```
class Pair
{
public:
    Pair() {}
    Pair(char firstValue, char secondValue)
        : first(firstValue), second(secondValue) {}
    char getFirst(){ return first;}
    . . .
private:
    char first;
    char second;
};
```

注意，在内联函数定义中，结束花括号之后不需要分号，但有分号也不算错。

编译器采取特殊方式处理内联函数定义，所以它们通常具有更高的执行效率，虽然也会消耗更多的内存。使用内联函数，程序中的每一次函数调用都会被替换成函数定义的一个已编译版本。所以，内联函数调用不会产生像普通函数调用那样的开销。

附录 F　重载数组索引方括号

可为类重载方括号[]，以便为类的对象使用它。要在赋值操作符左侧的表达式中使用[]，必须定义操作符来返回一个引用，这要求为返回类型添加符号&(这一点类似于重载 I/O 操作符<<和>>)。重载[]时，操作符[]必须是成员函数，而不能是友元操作符(在这一点上，[]的重载方式类似于对赋值操作符=的重载；=的重载问题在 11.4.5 节进行了讨论)。

例如，以下代码定义了一个名为 Pair 的类，它的对象具有字符数组的行为。“数组”中含有两个索引，分别是 1 和 2(而不是 0 和 1)：

```
class Pair
{
public:
    Pair();
    Pair(char firstValue, char secondValue);
    char& operator[](int index);
private:
    char first;
    char second;
};
```

成员函数[]可像下面这样定义：

```
char& Pair::operator[](int index)
{
    if (index == 1)
        return first;
    else if (index == 2)
        return second;
    else
    {
        cout << "Illegal index value.\n";
        exit(1);
    }
}
```

像下面这样声明和使用对象：

```
Pair a;
a[1] = 'A';
a[2] = 'B';
cout << a[1] << a[2] << endl;
```

注意，在 a[1]中，a 是调用对象，而 1 是传给成员函数[]的参数。

附录 G　this 指针

为类定义成员函数时，有时希望引用调用对象。this 指针是指向调用对象的预定义指针。例如下面这个类：

```
class Sample
{
public:
        . . .
    void showStuff();
        . . .
private:
    int stuff;
        . . .
};
```

下面是定义成员函数 showStuff 的两种方式，它们是等价的：

```
void Sample::showStuff()
{
    cout << stuff;
}

// 不是好的风格，但确实演示了 this 指针的用法
void Sample::showStuff()
{
    cout << (this->stuff)
}
```

注意，this 不是调用对象的名称，而是指向调用对象的指针的名称。this 指针的值不能改变；它总是指向调用对象。

正如上例的注释所示，一般都没有必要使用 this 指针。但它在少数几种情况下确实能带来方便。

经常在重载赋值操作符=时使用 this 指针。例如下面这个类：

```
class StringClass
{
public:
    . . .
    StringClass& operator=(const StringClass& right_side);
    . . .
private:
    char *a; // 为以'\0'结尾的字符串值准备的动态数组
};
```

借助后续的重载赋值操作符定义，可以写出下面这样的赋值链：

```
s1 = s2 = s3;
```

这个赋值链等价于以下形式：

```
s1 = (s2 = s3);
```

重载的赋值操作符的定义使用 this 指针返回操作符=左侧的对象(即调用对象)：

```
// 这个版本不支持所有情况，请参见下一个版本
StringClass& StringClass::operator=(const StringClass& right_ side)
{
    delete [] a;
    a = new char[strlen(right_side.a) + 1];
    strcpy(a, right_side.a);
    return *this;
}
```

上述定义在一种情况下会出问题：如赋值操作符两侧出现同一个对象(比如 s = s;)，那么数组成员会被删除。为避免这个问题，可用 this 指针测试这种特殊情况。如下所示：

```
// 修正 bug 之后的最终版本
StringClass& StringClass::operator=(const StringClass& right_side)
{
    if (this == &right_side)
    {
        return *this;
    }
    else
    {
        delete [] a;
        a = new char[strlen(right_side.a) + 1];
        strcpy(a, right_side.a);
        return *this;
    }
}
```

11.4 节为名为 StringVar 的类重载了赋值操作符。那一节不需要使用 this 指针，因为当时有一个名为 maxLength 的成员变量，可用它检测是否在赋值操作符=的两侧使用了同一个对象。使用前面讨论的 StringClass 类时，则无法借助这种辅助手段，因为只有一个成员变量。这种情况只能使用 this 指针。

附录 H　将操作符重载为成员操作符

本书在重载操作符时，通常将它们视为类的友元。例如第 11 章的图 11.5 将操作符+重载为友元。为此，需要在类定义中将操作符标记为 friend，如下所示：

```
// 用于表示美元金额的类
class Money
{
public:
    friend Money operator +(const Money& amount1, const Money& amount2);
        . . .
```

接着，在类定义的外部定义了重载的操作符+(参见图 11.5)。

其实还可将操作符+(和其他操作符)重载为**成员操作符**。要将操作符+重载为成员操作符，类定义要像下面这样开始：

```
// 用于表示美元金额的类
class Money
{
public:
    Money operator +(const Money& amount2);
```

注意，将二元操作符重载为成员操作符时只有一个参数(而不是两个)。调用对象作为第一个参数使用。例如以下代码：

```
Money cost(1, 50), tax(0, 15), total;
total = cost + tax;
```

其中，操作符+已重载为成员操作符，所以在表达式 cost + tax 中，变量 cost 就是调用对象，而 tax 是传给操作符+的参数值。

成员操作符+的定义如下：

```
Money Money::operator +(const Money& amount2)
{
    Money temp;
    temp.allCents = allCents + amount2.allCents;
    return temp;
}
```

在成员操作符的定义中，尤其要注意下面这行代码：

```
temp.allCents = allCents + amount2.allCents;
```

传给操作符+的第一个参数值是未限定的 allCents，所以是调用对象的成员变量 allCents。

将操作符重载为成员操作符刚开始可能觉得有些“别扭”，但习惯就好。许多专家提倡总是将操作符重载为成员操作符，而不要重载为友元。这更符合面向对象编程的精神。不过，将二元操作符重载为成员操作符，也存在一个巨大的不利因素。在这种情况下，两

个参数不再是对称的。一个是调用对象，只有第二个“参数”才是真正的参数值。这使代码不再“优雅”。此外，它还有一处非常现实的不足。任何自动类型转换都只应用于第二个参数。例如，以下语句是合法的：

```
Money baseAmount(100, 60), fullAmount;
fullAmount = baseAmount + 25;
```

这是由于 Money 含有接收单个 long 类型参数的一个构造函数，所以值 25 会被视为 long 值，所以会自动转换成 Money 类型的一个值。

然而，将+重载为成员操作符，则不能反转+的两个参数。以下语句非法：

```
fullAmount = 25 + baseAmount;
```

这是由于 25 不能成为调用对象。long 值到 Money 类型的转换只适用于参数值，不适用于调用对象。

另一方面，如将+重载为友元，则以下语句完全合法：

```
fullAmount = 25 + baseAmount;
```